UNIVERSITÄT OSNABRÜCK

AG Kombinatorische Optimierung

Bachelor thesis

Genetic Algorithms for the Traveling Salesman Problem

Valeriya Bechberger

June 10, 2020

First reviewer:	Prof. Dr. Sigrid Knust
Second reviewer:	Sven Boge M. Sc.

Zusammenfassung

Diese Bachelorarbeit wendet genetische Algorithmen auf das Traveling Salesman Problem an. Das Traveling Salesman Problem (TSP) ist ein NP-schweres Optimierungsproblem, bei dem die kürzeste Rundtour durch eine gegebene Menge an Städten gesucht wird. Genetische Algorithmen sind evolutionäre Algorithmen, die vom Prozess der natürlichen Selektion inspiriert sind. Diese Bachelorarbeit gibt einen Überblick über verschiedene Varianten genetischer Algorithmen im Zusammenhang mit dem TSP. Sie beschreibt die wichtigsten Repräsentationstypen für Rundtouren und diskutiert jeweils verschiedene Selektions-, Crossover- und Mutations-Operatoren. Außerdem analysiert diese Bachelorarbeit experimentell, wie effektiv genetische Algorithmen das TSP lösen. Zunächst werden die generellen Hyperparameter Populationsgröße und Mutationsrate optimiert. Im Hauptteil der Experimente werden dann die Ergebnisse verschiedener Repräsentationstypen sowie verschiedener Kombinationen von Crossover- und Mutationsoperatoren verglichen. Abschließend listet diese Bachelorarbeit die Operatoren mit der besten und schlechtesten Performance und vergleicht die beobachteten Ergebnisse mit den Effekten, die in der Literatur beschrieben wurden.

Abstract

This bachelor thesis applies genetic algorithms to the traveling salesman problem. The traveling salesman problem (TSP) is an NP-hard optimization problem which looks for a shortest round tour through a given set of cities. Genetic algorithms are evolutionary algorithms inspired by the process of natural selection. This thesis provides an overview of different variations of genetic algorithms for solving the TSP. It describes the main approaches for representing a round tour and discusses different variants of selection, crossover and mutation operators, respectively. Moreover, this thesis analyzes experimentally the effectiveness of genetic algorithms with respect to solving the TSP. First, the general hyperparameters of population size and mutation rate are optimized. In the main part of the experiments, the performance of different representation types and combinations of different crossover and mutation operators are compared to each other. Finally, this thesis reports the best and the worst performing operators and compares the observed results to effects described in the literature.

Contents

1 Introduction

The traveling salesman problem (TSP) is a well-known optimization problem, where one aims to identify a shortest round tour through a given set of cities, provided that each city must be visited exactly once. Despite the fact that the problem can be easily formulated, it is known to be NP-hard. This means that an exact solution cannot be found in polynomial time unless P = NP. Applying exact algorithms therefore requires enormous computing efforts. Nevertheless, the TSP is widely studied in the field of combinatorial optimization due to its great theoretical and practical importance. On the one hand, as one of the standard optimization problems it serves as a basis for studying general optimization methods. Successful methods can then be transferred to other optimization problems. On the other hand, the TSP can be applied to a wide variety of real-world problems in different fields which include of course transportation and logistics applications. The simplicity of the problem definition has however also caused many interesting applications in other areas, such as genome sequencing, scan chains, drilling problems, aiming telescopes and X-rays, data clustering, and locating power cables (see [2]).

There exist many methods to solve the TSP. However, being NP-hard, this problem requires a compromise between the quality of solutions and processing time. Different approximation techniques can be applied to find a reasonably good but not necessarily optimal solution. One of those techniques are so-called genetic algorithms.

Genetic algorithms are search and optimization algorithms inspired by natural selection and genetics. A genetic algorithm includes several common steps, including selection, crossover, and mutation. However, these steps can vary greatly among different approaches in their formulation. This gives rise to plenty of modifications and a great variability for this type of optimization method.

This bachelor thesis considers genetic algorithms as an optimization method for solving the TSP. The purpose of this thesis is twofold: On the one hand, it gives a detailed theoretical overview of genetic algorithms in context of the TSP. This overview is mainly based on the paper "Genetic algorithms for the traveling salesman problem" [21] by Jean-Yves Potvin. On the other hand, this thesis provides a unified implementation of genetic algorithms for the TSP. This implementation is used for a comparative analysis of different variations of genetic algorithms for solving the TSP. The existing literature in this field in general comes without an implementation and typically leaves many hyperparameters unspecified. Even if they are specified, "the results cannot be easily compared because various algorithmic designs and parameter setting are used" [21]. Therefore, our analysis can provide additional insight for comparing different proposals.

The remainder of this thesis is structured as follows: Chapter 2 gives some background with respect to the TSP and genetic algorithms. Chapter 3 summarizes different types for representing candidate solutions and specifies which ones were used for the current research. Chapter 4 introduces several heuristics which were used to construct the initial population in the experiments. Chapters 5, 6, and 7 give an overview of the existing selection, crossover, and mutation operators, respectively. After that, Chapter 8 reports experiments for finding good settings for the general parameters of genetic algorithms. Chapter 9 then contains a comparative study of different combinations of crossover and mutation operators for the individual representation types. Chapter 10 compares the effectiveness of different approaches for representing candidate solutions. Moreover, it reports the experiments for the operators which have shown the best performance. Finally, Chapter 11 concludes this thesis by summarizing the main findings and giving an outlook on future work.

2 General Definitions

This chapter provides a formal definition of the Traveling Salesman Problem (TSP) and gives an overview of existing approaches to solve it. Moreover, it introduces genetic algorithms with their main phases and discusses how they can be used to solve the TSP.

2.1 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is the problem where a salesperson needs to visit customers which are located in different cities and looks for a shortest round trip to fulfill this issue as defined by Hoos and Stützle (see [13]).

The TSP is typically modeled using a directed, edge-weighted graph $K_n = (V, E)$ (see [13], [14]), where each node $v \in V$ represents one of n cities (i.e., $|V| = n$). An edge $e = (u, v) \in E$ between two nodes $u, v \in V$ represents a connection between the corresponding cities, and the weight on this edge represents the distance between them. These edge weights are represented by the cost function $c : E \to \mathbb{Z}$.

In our implementation, the cost function is represented by a distance matrix D, where entries d_{ij} represent the distance $d(i, j)$ between cities i and j. That is, the cost function c is defined as $c(i, j) := d_{ij}$. If D is *symmetric*, that is, for all $i, j \in V : d_{ij} = d_{ji}$, the corresponding TSP instance is called *symmetric* as well. If D is not symmetric, than the corresponding TSP instance is called *asymmetric*. D is said to satisfy the triangle inequality if and only if for all $i, j, k \in V : d_{ij} + d_{jk} \geq d_{ik}$. This occurs in case of Euclidean TSP instances with V as a set of points in $\mathbb{R}^2$ and d_{ij} as the length of the straight line segment between i and j as defined by Laporte (see [15]). Note that in our experiments, we only consider symmetric Euclidean TSP instances.

We call two nodes *adjacent* if they are connected by an edge. Two edges are called *incident* if they share a node. Also, a node and an edge are called *incident* if the edge starts or ends at this node.

A round tour through all cities is represented by a cyclic permutation $\pi = (\pi_1, ..., \pi_n)$ of vertices, where $\pi_{n+1} := \pi_1$ (see [14]). Such a cyclic permutation is also known as a *Hamiltonian cycle*. An optimal solution for the TSP consists of a cyclic permutation π which minimizes the overall costs $c(\pi) := \sum_{i=1}^{n} c(\pi_i, \pi_{i+1})$. An example of a symmetric TSP instance in the form of a graph is given in Figure 1. One can use different start cities and visit the cities in two directions as the given instance is symmetric. This results in eight tours which correspond to the same optimal solution. Without loss of generality, we take the city 0 as a start city. As the instance is symmetric, we can go into two directions getting optimal tours $\pi_1 = (0, 2, 3, 1)$ and $\pi_2 = (0, 1, 3, 2)$.

Generally speaking, there are two main ways to solve the TSP, namely exact and approximate algorithms. Exact algorithms basically consider all possible solutions in order to choose an optimal one, resulting in enormous computational costs. The most direct approach would be to try all possible cyclic permutations. For a fully connected

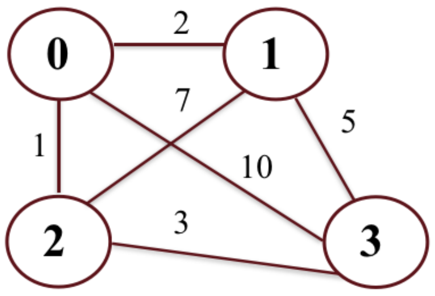


Figure 1: An example of a symmetric TSP instance with four cities.

graph with n nodes, their number is $(n-1)!$, resulting in the runtime $O(n!)$, the factorial of the number of cities. This growth in runtime, caused by increasing the number of cities n, is even faster than the exponential growth, as $O(n!) \supset O(2^n)$. This makes the problem of finding optimal round tour quite challenging. Of course, there are some approaches which improve the runtime, as, for instance, the dynamic programming approach proposed by Bellman [6] with the runtime $O(n^2 2^n)$. However, the growth in runtime still remains exponential. Therefore, running an exact algorithm for hours on a powerful computer may not be very cost-effective. In terms of quality and speed, solving the problem with approximate algorithms (also called heuristics) can be more feasible.

Approximate algorithms can be classified into three main types, namely:

- Construction heuristics
- Improvement heuristics
- Composite heuristics

Construction procedures start from an initial partial tour (usually a randomly chosen city) and iteratively extend it until a complete round tour has been constructed. A more detailed description of them is given in Chapter 4.

Improvement procedures start from a complete tour and try to reduce its length while ensuring that the result is still a valid tour. Gendreau and Potvin [9] have divided them further into *single-solution heuristics* and *population heuristics*. While the first group works with a single solution at a time, population heuristics work with multiple solutions evolving during the search process. One of the representatives of the latter group, namely genetic algorithms, will be discussed in the following section.

At last, **composite** approaches can be seen as a combination of construction and improvement procedures: They begin with a tour construction procedure to build an initial tour which is then refined using tour improvement procedures.

The classification of the above mentioned approaches is given in Figure 2 provided with some examples for each type.

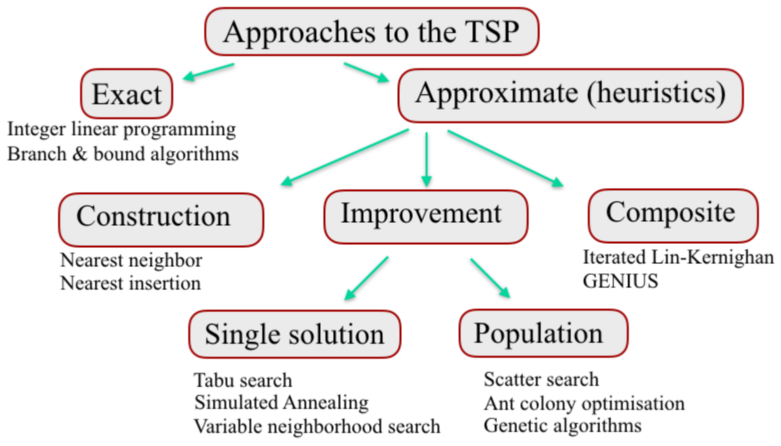


Figure 2: Existing approaches to solve the TSP (based on classifications given by Potvin [21], Gendreau and Potvin [9]).

2.2 Genetic Algorithms

Being a widely known metaheuristic, genetic algorithms are an approach for a randomized search introduced by Holland [12] which is based on the Darwinian principle of natural selection. Feasible solutions (usually encapsulated in the form of "chromosomes") build a population of a chosen finite size. A so-called fitness function assigns to each chromosome a fitness value which represents the quality of the corresponding solution. The fittest chromosomes of the population are more likely to be chosen by selection operators to produce offspring for the next generation. These new offspring chromosomes are created by applying crossover operators (which combine genetic material from two parents) and mutation operators (which produce small random perturbations with a given probability). Starting from a randomly or heuristically generated initial population, this process repeats until the defined stop criteria are reached. This general formulation of genetic algorithms is inspired by the description provided by Potvin [21] and Gendreau and Potvin [9]. It is illustrated in Figure 3.

This is a genetic algorithm template, where each phase is defined in a rather abstract way. In a specific domain, there is a number of open questions which have to be answered. First of all, which type of representation should be used? Each domain can have its unique types which have to be considered. How should the population be initialized? There are two main options to do that, namely creating chromosomes randomly or initializing them by applying heuristics. Another question to be answered is whether the size of the population remains fixed or whether the population can grow dynamically. In the first case, it is necessary to define this fixed size and in the latter case, one needs to specify to which extent the population can grow.

Another important decision concerns the choice of stop criteria for the algorithm. According to Safe et al. [24], there are three termination conditions which are typically used:

- maximum number of iterations (generations),

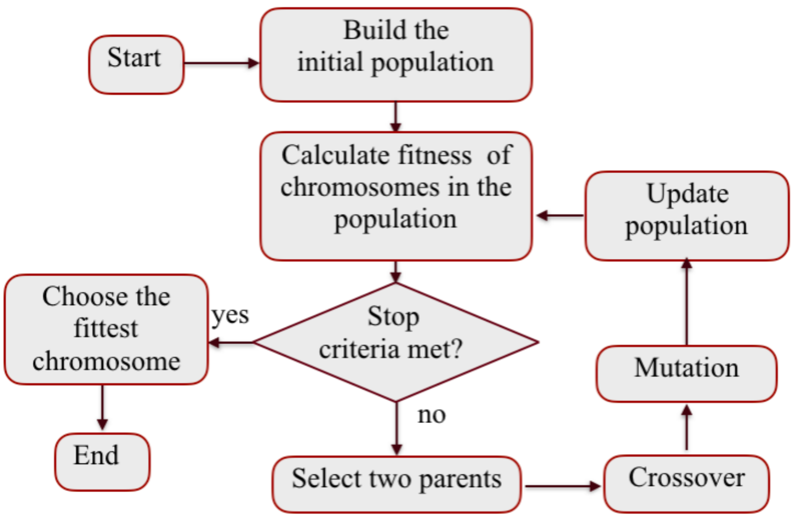

Start
Build the initial population
Calculate fitness of chromosomes in the population
Update population
Choose the fittest chromosome
yes
Stop criteria met?
Mutation
no
End
Select two parents
Crossover

Figure 3: The general procedure of an genetic algorithm.

- maximum number of evaluations of the fitness function, and
- a low chance of getting significant changes in the next generations.

In our experiments, we are going to stop the algorithm after a fixed amount of time. When comparing the results of different setups, we can thus estimate the quality of achieved results per unit of time which can be relevant for practical applications.

Further degrees of freedom involve the choice of selection, crossover and mutation operators. Different variants of these operators in the context of the TSP will be introduced in Chapters 5, 6, and 7, respectively. Please note that mutation operators are typically used with a certain probability. This probability is another parameter that needs to be determined.

Moreover, a decision needs to be made about the way of updating the population: Newly generated offspring chromosomes can replace their parents or not. In the former case, one also needs to specify under which conditions the replacements occur. One more question is whether mutations can be applied to only the offsprings which are produced through crossover, or to any chromosome from the population.

The answers to all these questions tend to be domain specific; therefore, they can be addressed using the best practices in the corresponding field. However, this domain knowledge is typically not sufficient for determining all choices. Thus, the unanswered questions can be considered as general parameters for the algorithm which then have to be optimized in order to find the best configuration.

From now on, we are going to focus on genetic algorithms for solving the TSP. For this purpose, some specifications have to be made. A feasible solution to a TSP problem is a cyclic permutation of the cities, or in other words, a feasible round tour, where each city appears exactly once. There are different ways to represent such a tour which will be discussed in Chapter 3. These round tours will be encoded in form of chromosomes which together make up the population. Distances between cities (which in our implementation are given in the form of a distance matrix) serve as a basis for the fitness function. The calculation of the fitness value of a chromosome is based on the length of the round tour it encodes. Please note that as we look for a round tour with minimal length, we need to minimize the whole distance of the tour. In our implementation, the fitness value of a tour corresponds to the length of this tour multiplied by minus one so that we can maximize the fitness function. In this case, the chromosome with the largest fitness value is considered to be the fittest and, correspondingly, the best one.

3 Representation Types

This chapter gives an overview of different ways for representing a tour of a TSP instance. We furthermore define which representation types will be used in our experiments. There are different possible representations for a tour of a TSP instance, namely:

- Binary representation
- Matrix representation
- Path representation (Section 3.1)
- Ordinal representation (Section 3.2)
- Random-key representation (Section 3.3)
- Adjacency representation (Section 3.4)

According to Larranaga et al. [16], the first two options are a standard way of representation used in genetic algorithms. However, in case of the TSP, applying crossover and mutation operators on them does not guarantee to result in valid tours. Therefore, they have been excluded from our subsequent considerations.

3.1 Path Representation

The path representation is the most natural representation of a tour. For example, a tour $\pi = (0, 1, 2, 3, 4)$ is represented by the path $0 \to 1 \to 2 \to 3 \to 4 \to 0$. From now on, we will use the shorthand notation 01234 for examples with less than ten nodes for this representation type, where it is implicitly assumed that there exists an edge from the last city to the first one. This representation leads to several chromosomes which represent the same tour because the start city can be different. Therefore, all these chromosomes have the same fitness value and can thus be considered as equivalent.

Note that this representation for a TSP tour can provoke problems when applying classical operators of genetic algorithms. For instance, the classical one-point crossover takes one cut point and defines the offspring by taking the part before the cut point from the first parent and the part after the cut point from the second parent. When using the path representation, the result is not necessary a valid tour. For instance, a one-point crossover for the chromosomes 051243 and 541203 may result in the offspring 051203 (see Figure 4). However, city 0 appears twice in the offspring while city 4 does not appear at all. It is obvious that the offspring does not represent a valid tour. This representation therefore requires additional measures to avoid such duplicates. Nevertheless, the simplicity of this representation has caused its large applicability. Moreover, there is a large number of operators for the path representation which ensure the validity of the offspring (see Section 6.2). For this reason, we will use the path representation in our experiments.

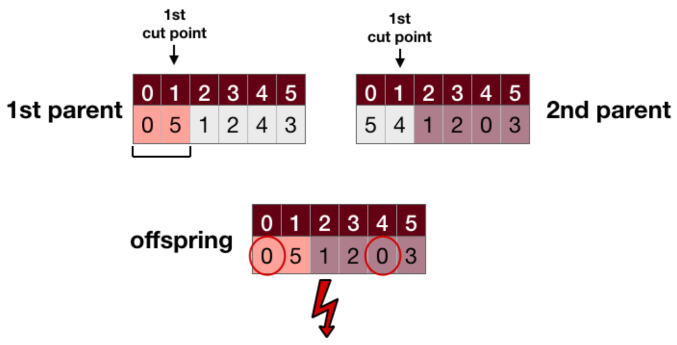


Figure 4: One-point crossover for chromosomes 051243 and 541203 in the path representation.

3.2 Ordinal Representation

The ordinal representation was proposed by Grefenstette et al. [11]. A tour in the ordinal representation is given as a list of n cities. Moreover, there exists a so-called "reference tour" which represents an ordered list of cities to decode and encode the tour in this representation. In order to get an ordinal representation, one has to go through the given tour and identify the position which the next city occupies in the reference tour. This index is used as the next element in the ordinal representation. After that, this city is deleted from the reference tour. An analogous procedure can be used to decode the ordinal representation.

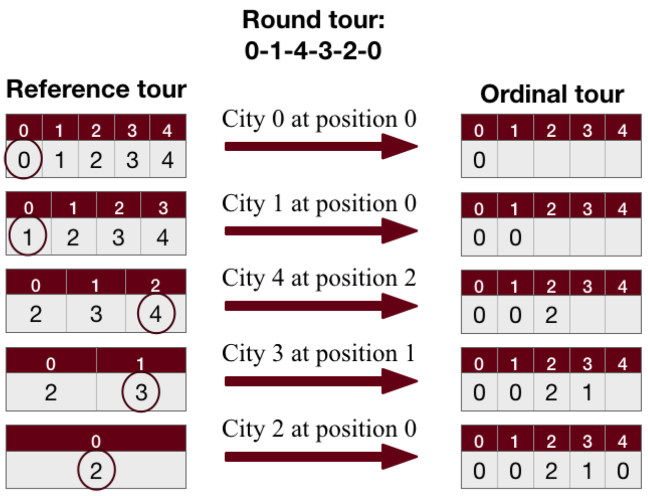


Figure 5: Ordinal representation for a tour $\pi = (0, 1, 4, 3, 2)$.

For instance, let us consider the tour $\pi = (0, 1, 4, 3, 2)$ (see Figure 5). We use the ascendingly ordered list of cities as reference tour $ref = (0, 1, 2, 3, 4)$. To get an ordinal representation, we have to go through the cities of π and define their positions in ref. The first city of the tour is the city 0 which occupies the 0th position in the reference tour. We delete this city from the reference tour, getting $ref = (1, 2, 3, 4)$ and $ordinal = (0)$. The next city of tour π is city 1 which stands now at the position 0 in the reference tour; therefore, 0 is added to the ordinal representation. After deleting city 1 from the reference tour, we get $ref = (2, 3, 4)$ and $ordinal = (0, 0)$. City 4 is the next one and it takes position 2 in the reference tour. Thus, we get $ref = (2, 3)$ and $ordinal = (0, 0, 2)$. City 3 has position 1, resulting in $ref = (2)$ and $ordinal = (0, 0, 2, 1)$. The last city in π is city 2 which stands at the position 0 of the reference tour. Therefore, 0 is added to the ordinal representation, and we finally get $ref = ()$ and $ordinal = (0, 0, 2, 1, 0)$.

We will now show how the decoding procedure works. Having the ordinal tour $ordinal = (0, 0, 2, 1, 0)$ and the reference tour $ref = (0, 1, 2, 3, 4)$, the first number in the ordinal representation is 0. This means that if we want to get the first city of the original tour, we need to get the element in the reference tour which stands at the position 0 and remove it from the reference tour after that. So, we get the city 0, obtaining the partial tour $\pi = (0)$ and $ref = (1, 2, 3, 4)$. The next element in the ordinal representation is 0 as well. This means that we have to look for a city again at the position 0 in the reference tour. Now it is city 1. As a result, we get the partial tour $\pi = (0, 1)$ and $ref = (2, 3, 4)$. The next element of the ordinal tour, namely 2, brings us to the city 4 which stands at this position in the reference tour, resulting in the partial tour $\pi = (0, 1, 4)$ and $ref = (2, 3)$. After that, we find the city 3 at the position 1, leaving us with $\pi = (0, 1, 4, 3)$ and $ref = (2)$. The last element of the ordinal tour is 0. This will always be the case because the reference tour only consists of a single element now. City 2 is this remaining city in the reference tour, and it finishes the round tour π. Thus, the original tour is $\pi = (0, 1, 4, 3, 2)$.

Potvin [21] showed that if a classical one-point crossover (which did not work for the path representation) is applied in this representation, the generated offspring will always represent a valid tour. He explained it by the fact that each value in the ordinal tour corresponds to a particular position in the reference tour. While applying a one-point crossover, exchanging values between two parent chromosomes simply modifies the order of selection of the cities in the reference tour. Therefore, a valid permutation is always generated. Larranaga et al. [16] pointed out that the partial tours to the left of the cut point do not change, whereas the partial tours to the right of it are disrupted in a quite a random way. According to Larranaga et al. [16], these disruptions are the reason for relative poor results which this representation type has shown. Also Potvin [21] has also pointed out that this representation type can be considered mainly out of historical interest because the sequences of cities in the parents are not well inherited in the offspring, resulting in mostly random permutations. We are going to consider this representation type in our experiments to find out whether it shows bad performance on our data set as well.

3.3 Random-Key Representation

The concept of *random keys* for representing round tours was proposed by Bean [3]. This representation uses random numbers typically drawn uniformly from the interval $[0, 1)$. Each chromosome consists of a list of such randomly drawn numbers. It is assumed that the ith random number is associated with city i. By sorting the chromosomes in ascending [3] or descending [14] order, one can thus obtain a valid permutation of the cities.

For instance, let us assume that we have drawn the following random keys which encode a tour of length 5 : $keys = (0.47, 0.05, 0.34, 0.99, 0.55)$. Now, we assign these random numbers to the cities, namely: $0.47 \rightarrow 0, 0.05 \rightarrow 1, 0.34 \rightarrow 2, 0.99 \rightarrow 3, 0.55 \rightarrow 4$. Then, we sort the keys in ascending order and get: *sorted* =(0.05(1), 0.34(2), 0.47(0), 0.55(4), 0.99(3)). Therefore, the decoded tour is $\pi = (1, 2, 0, 4, 3)$.

According to Snyder and Daskin [26], the random-key representation is useful for problems that require permutations of integers where traditional crossovers have problems with producing valid offsprings (as, for instance, a one-point crossover in the path representation, as described above in Section 3.1). Standard crossover techniques applied in this representation will generate children that are guaranteed to be valid.

Nevertheless, in our implementation, we do not work with the random-key representation because we use construction heuristics to produce chromosomes for the initial population. These heuristics return round tours in the path representation. There is, however, an infinite number of ways to translate a tour from the path representation into the random-key representation: We need to ensure that the keys are ordered in the correct way, but the numerical value of the keys is not determined. The actual numeric value does, however, play a critical role when applying crossover and mutation operators in this representation. As it is unclear how to translate the initial population from the path representation to the random-key representation, we do not consider the latter in our experiments.

3.4 Adjacency Representation

In this representation type, each array index represents one city. If the value j is found at index i of the array, this means that the round tour contains an edge (i, j) from city i to city j. In the path representation, we have a designated starting point which results in n chromosomes representing the same round tour. In contrast to that, there is no designated starting point in the adjacency representation, but this representation type is directed. For symmetric TSP instances, the same round tour which goes in one direction will look different from the one which goes in the opposite direction. Therefore, each round tour can be mapped to two chromosomes for symmetric TSP instances and exactly to one chromosome for asymmetric TSP instances in adjacency representation.

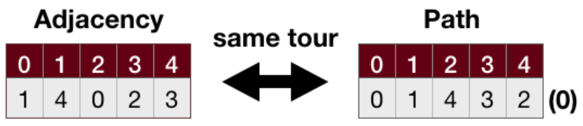


Figure 6: The same round tour in path and adjacency representation.

One can easily transform a tour in this representation into the path representation (see Figure 6): City 0 is always added to the path tour at the beginning automatically.

Then, one starts at the index 0 and looks which city stands at this index in the adjacency representation. This city becomes the next city of the path tour. After that, this city is used as the next index and the city which takes this index continues the path tour. The process repeats until the whole path is build. For instance, consider the tour 14023 in the adjacency representation. In the adjacency representation, we find city 1 at the index 0 which means that city 0 is followed by city 1 in the round tour. We thus append it to our path. At the index 1, stands city 4, which is added to the path. At the index 4, we find city 3. Its neighbor is city 2, as it stands at the index 3. City 0 is at the index 2 which completes the Hamilton cycle 01432(0) in the path representation.

An inverse transformation from the path representation into the adjacency representation is possible as well. One goes along the path tour and for each pair of neighbors π_i and π_{i+1} in the path tour $\pi = (\pi_1, ..., \pi_i, \pi_{i+1}, ...\pi_n)$, city π_{i+1} takes index π_i in the adjacency representation. For instance, let us reuse the example in Figure 6. We have the tour 01432 in the path representation. The first pair of cities includes 0 and 1. This means that city 1 takes position 0. The next pair contains cities 1 and 4, resulting in that city 4 takes position 1 in the adjacency representation. In the same way, city 3 takes position 4, and city 2 takes position 3. As a round tour is needed in case of TSP instances, we need to make a connection between city 2 and 0. Therefore, city 0 stands at the position 2 and finishes the tour.

Please note that like the path representation also the adjacency representation needs special operators to preserve the validity of the chromosomes (see Section 6.3). We included this representation type with its specific crossover operators in our implementation to compare its effectiveness with the one of the path representation. As a translation from the path representation to the adjacency representation is quite straightforward, we can also use the tours found by the construction heuristics with this representation type.

4 Constructive Heuristics

One of the first important steps in a genetic algorithm is to construct an initial population (see Section 2.2). This can be done by generating solutions randomly or by using heuristics. The latter approach causes additional computational costs, but it guarantees that the first generation has at least several reasonably good solutions.

This chapter gives an overview of heuristics [14] which will be used in our research as a basis to construct an initial population for genetic algorithms. We have chosen four heuristics which are all of the constructive type because construction heuristics find feasible solutions in relatively short time. Please note that the generated solutions are not guaranteed to be optimal – if they were, we would not need to apply a genetic algorithm.

Constructive heuristics build the tour successively and greedily by using deterministic construction rules without trying to improve the existing partial tour. We will use the term "partial tour" in the following to denote the part of the resulting tour which has already been built. Please note that it is important for these heuristics which city will be visited first because the solutions differ dependent on which starting point is chosen. Moreover, please note that the tour is given in the path representation here.

We will use the following four heuristics: nearest neighbor, double nearest neighbor, nearest insertion, and farthest insertion. They are described in detail below.

4.1 Nearest Neighbor

Considering a partial tour is already built, the nearest neighbor (NN) heuristic iteratively chooses a city which is closest to the last city of the tour and inserts it at the end.

The algorithm includes the following steps:

1. Choose an arbitrary city π_1 as a starting point for the current partial tour $\pi := (\pi_1)$.
2. While there are unvisited cities left:

   Given the partial tour $\pi := (\pi_1, ..., \pi_t)$, find the next unvisited city $\pi_k \notin \pi$ which is nearest to the city π_t and add it to the partial tour such that $\pi := (\pi_1, ..., \pi_t, \pi_k)$.

For instance, consider the five cities with symmetric distances from Table 1. Let us use city A as the start city, resulting in an initial partial tour $\pi = (A)$. The next unvisited city which has the smallest distance to city A is city C with a distance of 4. We add this city after city A getting a partial tour $\pi = (A, C)$. The nearest unvisited neighbor of city C is city D with the distance 5. Now the tour consists of three cities: $\pi = (A, C, D)$. The next unvisited city which is closest to city D is city E with a distance of 4. The partial tour is now $\pi = (A, C, D, E)$. City B is the last unused city having a distance of 10 to city E resulting in a tour $\pi = (A, C, D, E, B)$ (see Figure 7) with the total costs $c(\pi) = 32$ including the distance between city B and A (i.e., 8). If we start with city C, the NN heuristic results in the round tour $\pi' = (C, A, B, D, E)$ (see Figure 8) with the total costs $c(\pi') = 4+8+7+4+6 = 29$ which is shorter than the previous one. This illustrates the sensitivity of the construction heuristics to the start city.

	A	B	C	D	E
A	0	8	4	9	9
B	8	0	6	7	10
C	4	6	0	5	6
D	9	7	5	0	4
E	9	10	6	4	0

Table 1: Distance table for five cities.

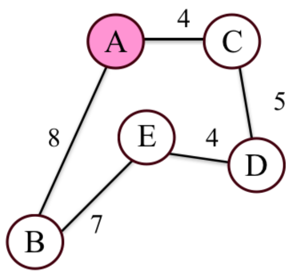


Figure 7: Nearest neighbor resulting tour for an example with five cities with distances in Table 1 and start city A.

4.2 Double Nearest Neighbor

The double nearest neighbor (DNN) heuristic is a modified version of the NN heuristic: It looks for a neighbor which is closest to either the first or the last city of the existing partial tour. This neighbor will be inserted before the first city or after the last city, respectively. The DNN algorithm includes the following steps:

1. Choose an arbitrary city π_1 as a starting point for the current partial tour $\pi := (\pi_1)$.
2. While there are unvisited cities left:

   a. Given the partial tour $\pi := (\pi_1, ..., \pi_t)$, find an unvisited city $\pi_i \notin \pi$ which is nearest to the first city π_1 and an unvisited city $\pi_j \notin \pi$ which is nearest to city π_t. Let $d(\pi_i, \pi_1)$ be the distance between π_i and π_1 and $d(\pi_t, \pi_j)$ be the distance between π_t and π_j.

   b. If $d(\pi_i, \pi_1) < d(\pi_t, \pi_j)$: Insert city π_i at the beginning of the partial tour such that $\pi := (\pi_i, \pi_1, ..., \pi_t)$.

   c. Otherwise, if $d(\pi_i, \pi_1) \geq d(\pi_t, \pi_j)$: Insert city π_j at the end of the partial tour such that $\pi := (\pi_1, ..., \pi_t, \pi_j)$.

Let us reuse the example with the five cities from Table 1. If we start with city A here, the DNN will result in the same round tour as the NN does. Therefore, we start with city C. Its nearest neighbor is city A with the distance 4, resulting in a partial tour $\pi = (C, A)$ Now we have to find unvisited cities which are nearest to cities A and C, respectively. City B is the nearest to city A with the distance 8, and city D is nearest to city C with a distance of 5. Therefore, we add city D at the beginning before city C which results in the partial tour $\pi = (D, C, A)$. The next unvisited city nearest to city D is city E with a distance of 4, while the nearest unvisited neighbor of city A is still city B. This means that city E will be added to the tour before city D resulting in $\pi = (E, D, C, A)$. The last city B will be added to the tour resulting in the round tour $\pi = (E, D, C, A, B)$. If we want the start city to be at the beginning of the tour, this can be written as $\pi = (C, A, B, E, D)$. The total costs are $c(\pi) = 4+8+10+4+5 = 31$. Figure 8 compares the tours found by the NN and the DNN heuristics, respectively, with city C used as starting point. As we can see, the DNN is not necessarily always better than the NN, although more computations have been made.

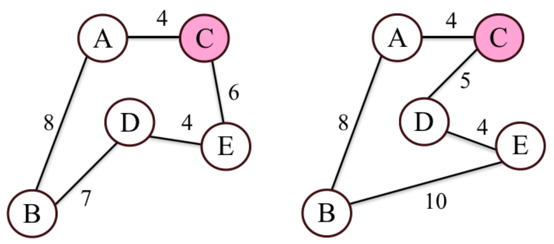


Figure 8: Tours found by the NN heuristic (left) with costs 29 and the DNN heuristic (right) with costs 31, both starting with city C for an example with five cities with distances from Table 1.

4.3 Nearest Insertion

Generally speaking, the nearest insertion algorithm (NI) looks iteratively for an unvisited city which is closest to *any* city from the partial tour and inserts this city in the tour at the position where the insertion costs are minimal. If considered in detail, the steps are the following:

1. Choose an arbitrary city π_1 as a starting point for the partial tour $\pi := (\pi_1)$.
2. While there are unvisited cities left:

   a. Let us assume that the partial tour is given as $\pi := (\pi_1, ..., \pi_t)$. For each unvisited city $\pi_j \notin \pi$ calculate the distance $d_\pi(\pi_j)$ to the partial tour π which is defined as $d_\pi(\pi_j) := \min_{i=1}^{t} d(\pi_j, \pi_i)$.

   b. Choose city $\pi_j^* \notin \pi$ with the smallest distance $d_\pi(\pi_j^*)$.

   c. Insert city π_j^* between those neighbor cities $\pi_l \in \pi$ and $\pi_{l+1} \in \pi$, where the insertion costs $c(\pi_l, \pi_j^*, \pi_{l+1}) := d(\pi_l, \pi_j^*) + d(\pi_j^*, \pi_{l+1}) - d(\pi_l, \pi_{l+1})$ are the smallest.

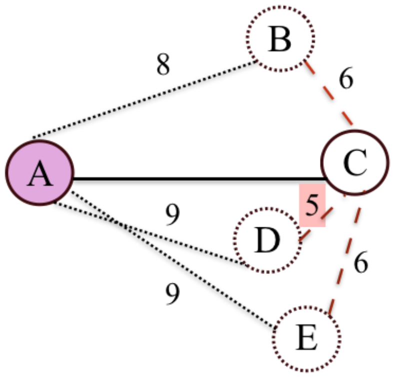


Figure 9: Distances to the partial tour $\pi = (A, C)$ for cities B, D, and E (on red dashed lines).

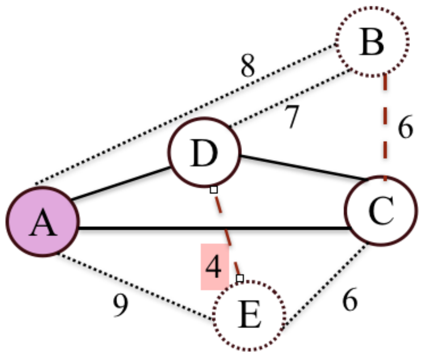


Figure 10: Distances to the partial tour $\pi = (A, D, C)$ for cities B and E (on red dashed lines).

Let us again consider the example with five cities with distances given in Table 1. We will start with city A giving us the initial partial tour $\pi = (A)$. As π consists only of one city, we look for an unvisited city nearest to city A which is city C. We add it to the partial tour π just after city A without calculating insertion costs, as there is only one option to insert at the moment. This results in the partial tour $\pi = (A, C)$. Now, for each unvisited city, we calculate its distance to the partial tour π: $d_\pi(B) = 6$, $d_\pi(D) = 5$, and $d_\pi(E) = 6$. City D has the smallest distance and will be chosen (see Figure 9). As there is only one pair of cities in the partial tour π, there is only one possible location for city D. This results in the partial tour $\pi = (A, D, C)$. The next unvisited city which has the smallest distance to the partial tour π is city E (see Figure 10). As there are 3 cities in the current partial tour π, there are three pairs between which city E could be inserted. The costs of inserting city E between cities A and D are $c(A, E, D) = 9 + 4 - 9 = 4$. Moreover, $c(D, E, C) = 4 + 6 - 5 = 5$ and $c(C, E, A) = 6 + 9 - 4 = 11$. Therefore, city E is inserted between cities A and D, resulting in $\pi = (A, E, D, C)$. City B is the last unvisited city. Its insertion costs are $c(A, B, E) = 8+10-9 = 9$, $c(E, B, D) = 10+7-4 = 13$, $c(D, B, C) = 7+6-5 = 8$, and $c(C, B, A) = 6+8-4 = 10$. Therefore, city B will be inserted between cities D and C, resulting in the tour $\pi = (A, E, D, B, C)$ with the total costs $c(\pi) = 9+4+7+6+4 = 30$. If the NN and DNN heuristics are started with city A as well, they both produce tours with costs 32. This illustrates that the NI heuristic can be useful.

4.4 Farthest Insertion

The farthest insertion (FI) heuristic is almost identical to the NI heuristic. It only differs from the NI by choosing a city among the list of unvisited cities whose minimal distance to the partial tour π is *maximal*. We can thus modify the NI algorithm by replacing line b with the following line b':

b'. Choose city $\pi_j^* \notin \pi$ with the largest distance $d_\pi(\pi_j^*)$.

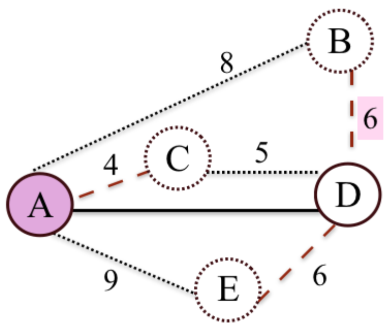


Figure 11: Distances to the partial tour $\pi = (A, D)$ for cities B, C, and E (on red dashed lines).

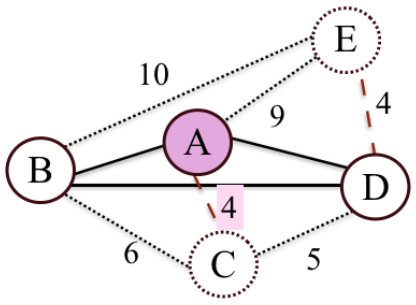


Figure 12: Distances to the partial tour $\pi = (A, B, D)$ for cities C and E (on red dashed lines).

While this modification may sound counter-intuitive, the following example illustrates that the FI heuristic can sometimes give better results than the NI one: Let us consider the same example with five cities and distances given in Table 1 with the start city A. Our initial partial tour is $\pi = (A)$. Cities D and E have the maximal distance $d_\pi(D) = d_\pi(E) = 9$ to the partial tour π. For cities with identical distance, we use the one appearing first in the alphabetical order, i.e. city D. This results immediately in the partial tour $\pi = (A, D)$, as there is only one option to insert city D. The next unvisited city with the largest distance d_π to the already build partial tour π is city B with a distance of 6 (see Figure 11). There is only one pair in π now; therefore, there exists only one option to insert city B. This results in the partial tour $\pi = (A, B, D)$.

Both remaining cities C and E have an identical distance $d_\pi(C) = d_\pi(E) = 4$ to this partial tour (see Figure 12). We choose city C based on the alphabetical order. The insertion costs for city C are $c(A, C, B) = 4+6-8 = 2$, $c(B, C, D) = 6+5-7 = 4$, and $c(D, C, A) = 5 + 4 - 9 = 0$. Therefore, city C will be inserted between cities D and A, resulting in $\pi = (A, B, D, C)$. The last unvisited city is city E. Its insertion costs are $c(A, E, B) = 9 + 10 - 8 = 11$, $c(B, E, D) = 10 + 4 - 7 = 7$, $c(D, E, C) = 4 + 6 - 5 = 5$, and $c(C, E, A) = 6 + 9 - 4 = 11$. Therefore, city E will be inserted between cities D and C, resulting in $\pi = (A, B, D, E, C)$ with total costs $c_\pi = 8 + 7 + 4 + 6 = 4 = 29$. This tour produced by the FI is thus shorter than the one produced by the NI which had a total distance of 30.

5 Selection Operators

This chapter gives an overview of different types of selection operators which are used to select chromosomes from the population for producing offspring. We consider the following selection operators, naming them according to the terminology used by Razali and Geraghty [22]:

- Random selection
- Proportional roulette wheel selection (Section 4.1)
- Rank-based roulette wheel selection (Section 4.2)
- Tournament selection (Section 4.3)

Random selection picks randomly two parents from the current population, where all chromosomes typically have the same probability of being selected (see [14]). This means that the fitness value of a chromosome has no influence on its probability of producing offspring. Therefore, random selection is quite straightforward but may not produce optimal results.

In contrast to that, the next three selection operators do consider the fitness values when selecting chromosomes. They will be explained in detail in the sections below.

5.1 Proportional Roulette Wheel Selection

In this type of selection, each chromosome has a probability of being selected which is directly proportional to its fitness value. This selection operator is called "roulette wheel" as the probabilities correspond to portions in a roulette wheel. In this way, chromosomes with a larger fitness value are more probable to be selected.

Let POP be the current population and f_i be the fitness value of the chromosome $i \in POP$. The selection probability p_i for this chromosome i is defined as $p_i := f_i / \sum_{j \in POP} f_j$, and the accumulated value is calculated as $q_i = \sum_{j=1}^{i} p_j$ (see [14]). To select a chromosome for reproduction, a random number $rand \in [0, 1]$ is drawn: If $rand \leq q_1$, the chromosome with the index 1 is selected. Otherwise, one chooses the chromosome with the index $i \geq 2$, where $q_{i-1} < rand \leq q_i$.

For instance, let us assume that a population contains 4 chromosomes with the fitness values 8, 4, 20, and 32, respectively. The sum of all fitness values is 64. Therefore, $p_1 = 8/64 = 0.125$, $p_2 = 4/64 = 0.0625$, $p_3 = 20/64 = 0.3125$, and $p_4 = 32/64 = 0.5$ (see Table 2 and Figure 13). Now the accumulated values have to be calculated: $q_1 = 0.125$, $q_2 = 0.125 + 0625 = 0.1875$, $q_3 = 0.1875 + 0.3125 = 0.5$, and $q_4 = 0.5 + 0.5 = 1$. If we draw $rand = 0.01$, the first chromosome is chosen, as $0.01 < 0.0625$. If, on the other hand, we draw $rand = 0.8$, then the chromosome with the index 4 is chosen, as $q_3 < 0.8 < q_4$.

Thus, the proportional roulette wheel selection gives a chance to all chromosomes in the population, assigning higher priority to the ones with larger fitness values. On the

Chromosome	*1*	*2*	*3*	*4*
Fitness value	*8*	*4*	*20*	*32*
Selection probability (p)	*0,125*	*0,0625*	*0,3125*	*0,5*
Accumulated value (q)	*0,125*	*0,1875*	*0,5*	*1*

Table 2: Selection probabilities and accumulated values for four chromosomes with fitness values 8, 4, 20, and 32, respectively, for the proportional roulette wheel selection (based on an example by Knust in [14]).

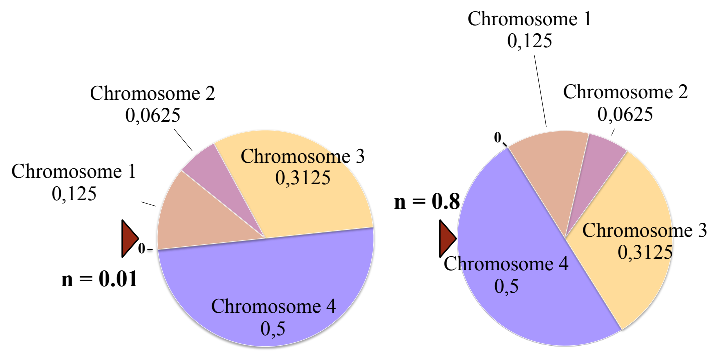


Figure 13: Selection probabilities in form of a "roulette wheel" for the four chromosomes from Table 2.

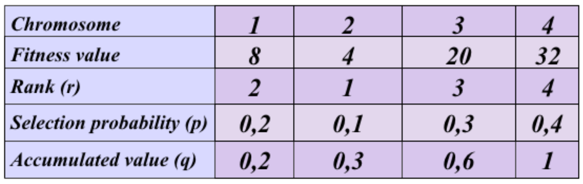
one hand, the fittest chromosomes are more likely to be selected. This corresponds to the idea of natural selection where the fittest individuals tend to survive. On the other hand, the other chromosomes can still be selected as well. However, Razali and Geraghty [22] pointed out that this selection type has a disadvantage: If an initial population contains a pair of very good (but still not optimal) chromosomes, and if the remainder of the population has relatively low fitness values, then these "good" chromosomes can quickly dominate the whole population. This may cause the genetic algorithm to get stuck in local minima.

Due to this shortcoming, other selection procedures have been developed, where the selection probability is not directly proportional to the fitness value of the chromosomes. They will be described in the following sections.

5.2 Rank-Based Roulette Wheel Selection

In the proportional roulette wheel selection, the selection probabilities are defined by the absolute fitness values of chromosomes. In contrast to that, the rank-based roulette wheel selection does not use fitness values for computing the selection probabilities directly. This selection type constructs a ranking of chromosomes instead: Each chromosome $i \in POP$ gets a rank value r_i, where the chromosome with the lowest fitness value gets the rank $r = 1$, and the chromosome with the largest fitness value obtains the rank $r = |POP|$. The selection probabilities p_i for each chromosome $i \in POP$ are calculated according to the defined ranks, namely $p_i = r_i / \sum_{j \in POP} r_j$ (see [14]). Please note that selection probabilities here can be also considered as portions in a roulette wheel; therefore, this selection type also contains "roulette wheel" in its name.

Let us again consider the example from Section 4.1, i.e. four chromosomes with fitness values 8, 4, 20, and 32, respectively. The computed ranks for these four chromosomes are: $r_1 = 2$, $r_2 = 1$, $r_3 = 3$, and $r_4 = 4$. This ranking results in the following selection probabilities: $p_1 = 2/(2+1+3+4) = 0.2$, $p_2 = 0.1$, $p_3 = 0.3$, and $p_2 = 0.4$ (see Table 3). In Figure 14, we compare the selection probabilities for proportional and rank-based roulette wheel selection for this example. Obviously, the chromosomes with lower fitness values have a higher chance of being selected in the rank-based variant than in the proportional one. This can be easily seen by comparing the two charts in Figure 14. For instance, the randomly drawn number $rand = 0.19$ results in selecting chromosome 3 in the proportional roulette wheel selection, whereas the rank-based variant still chooses the first chromosome.



Chromosome	*1*	*2*	*3*	*4*
Fitness value	*8*	*4*	*20*	*32*
Rank (r)	*2*	*1*	*3*	*4*
Selection probability (p)	*0,2*	*0,1*	*0,3*	*0,4*
Accumulated value (q)	*0,2*	*0,3*	*0,6*	*1*

Table 3: The selection probabilities and accumulated values for four chromosomes with the fitness values 8, 4, 20, and 32, respectively, for the rank-based selection.

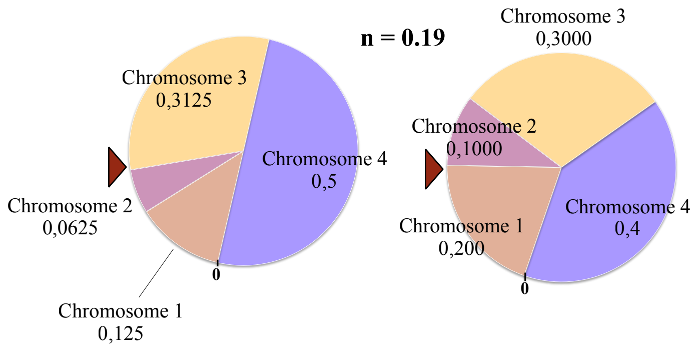


Figure 14: Selection probabilities for the proportional roulette wheel selection (left) and the rank-based roulette wheel selection (right) computed for four chromosomes with fitness values 8, 4, 20, and 32, respectively.

Therefore, this selection type solves the problem of getting stuck in local minima which can occur in the proportional roulette wheel selection because the influence of the chromosomes with very good fitness values is not so strong anymore. According to Razali and Geraghty [22], the main disadvantage of this approach is that the population has to be sorted for constructing the ranking which can result in high computational costs. Moreover, there can appear a problem of slow convergence because the selection probabilities of the best chromosomes do not differ so much from the ones of the other chromosomes.

5.3 Tournament Selection

The tournament selection considers p randomly chosen chromosomes from the population for selection. Among this group of chromosomes, the one with the highest fitness value is selected. The number of chromosomes p which take part in the tournament can depend on the size of the population.

One possible choice is to set p to a fixed constant. This is for example done in a so-called binary tournament which works with two participants (see [10]). Another option is to choose p randomly as well. When choosing p, one should consider two effects: On the one hand, if the number of participants is too large, chromosomes with smaller fitness values are selected very rarely because the best chromosomes always take part in the tournament. On the other hand, if p is too small in comparison to the population size, only very few chromosomes take part in the tournament. Especially if the number of generations is not large, this means that a large proportion of the population is never considered for selection.

In comparison to the previous two selection types, the tournament selection gives a greater chance for the chromosomes with low fitness values (if p is not large). The chromosomes are taken into the tournament independent from their fitness values. Therefore, the first advantage of the tournament selection is that it cannot be easily trapped in local minima. Secondly, the implementation is easier: It does not require additional calculations for probabilities as in the proportional roulette wheel selection and rank-based roulette wheel selection. Thirdly, we do not need to sort chromosomes like in rank-based selection; therefore, it is not so computationally costly. The main disadvantage of tournament selection is that the chromosomes with the best fitness values may not be selected at all, if the number of generations is not large and the number of participants in the tournament is small.

A number of surveys have been conducted to compare the performance of different selection strategies. Razali and Geraghty [22] compared the performance of three selection strategies (proportional roulette wheel, rank-based roulette wheel and tournament selection) in a genetic algorithm to solve the TSP. According to their results, the tournament selection strategy outperformed other selection types, achieving best solution quality with low computing times. Goldberg and Deb [10] and, much later, Sharma and Wadhwa [25] made a comparative analysis of selection schemes used in genetic algorithms in general. Also in their studies, the tournament selection showed one of the best results. We therefore decided to use this type of selection operator in our experiments.

In our implementation, we are going to use a relatively small number of participants to avoid local minima, while providing a high number of iterations such that a larger part of the chromosomes has a chance to be selected.

6 Crossover Operators

This chapter gives a general overview of the different variations of crossover operators. There exists a great number of different crossover operators which have been developed for different specific problems. In this chapter, we will concentrate not only on crossover operators specifically designed for the TSP but we will also discuss some other operators which were used in the literature for other optimization problems.

Section 6.1, introduces a classical one-point crossover (OPX). In Section 6.2, we introduce path representation crossovers and illustrate how they work on examples. Section 6.3 focuses on adjacency representation crossovers providing the examples of the corresponding approaches. Note that each example generates only one offspring. The second offspring can be generated by switching the roles of two parent chromosomes. Finally, Section 6.4 introduces the edge recombination crossover (ERX).

6.1 One-Point Crossover

One-point crossover (**OPX**) was developed by Holland [12] to work with chromosomes which were represented as bit strings. Two parent chromosomes are cut at a randomly chosen position. The part before the cut point is taken from the first parent chromosome while the part after the cut point is taken from the second parent chromosome.

For instance, we have the first parent chromosome 110100 and the second parent chromosome 000111. Let 1 be our randomly chosen cut point. This means that the substring 11 is taken from the first parent, and the substring 0111 is taken from the second parent. The resulting offspring is 110111 (see Figure 15).

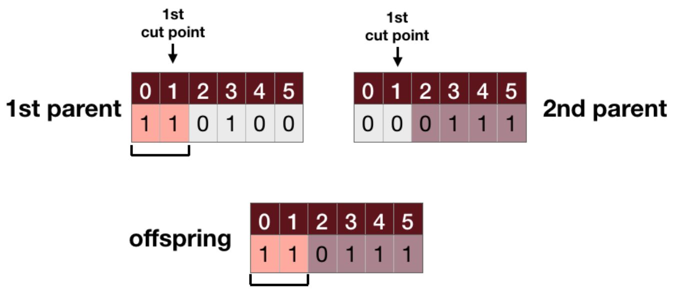


Figure 15: One-point crossover between parent chromosomes 110100 and 000111 in bit-string representation.

This crossover operator always produces valid offsprings for the TSP only in the ordinal representation which was pointed out in Section 3.2. For instance, let 02000 be the first parent chromosome in the ordinal representation which corresponds to 03124 in the path representation. Then, let 00210 be the second parent chromosome in the ordinal representation which corresponds to 01432 in the path representation. Let 1 be a randomly chosen cut point. The resulting offspring is 02210 which corresponds to 03421 in the path representation (see Figure 16).

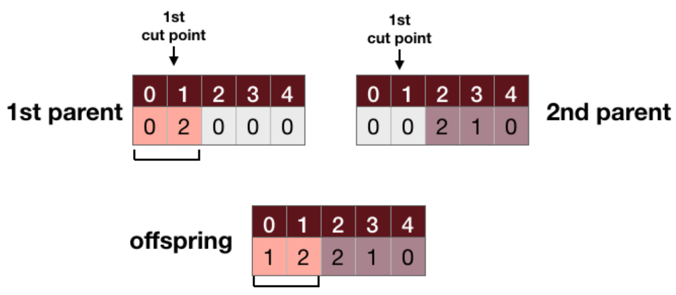


Figure 16: One-point crossover between parent chromosomes 02000 and 00210 in ordinal representation.

However, this crossover operator cannot be used directly for the TSP in the path and adjacency representation because simply exchanging parts of the parent chromosomes mostly results in invalid offsprings (see Figure 17). Therefore, some transformations of this operator are required, where the validity of the offspring is guaranteed. As a result, a great number of other crossover operators has been proposed which achieve this goal. We are going to discuss them in the following sections.

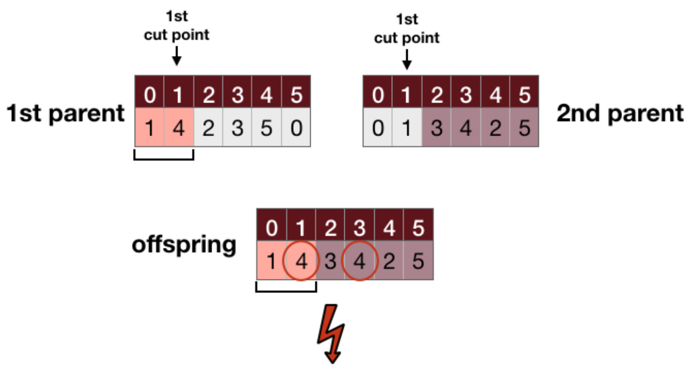


Figure 17: One-point crossover between parent chromosomes 142350 and 013425 in path representation for the TSP.

6.2 Path Representation Crossovers

The crossover operators of this type work with the chromosomes which encapsulate a tour in the path representation. These crossover operators differ mainly in the way of preserving the order of the cities from the parent chromosomes. Some of them preserve the absolute positions and the others preserve the relative order. The crossover operators of these group are:

- Crossover operators preserving absolute positions:

  Partially-mapped crossover,

  Cycle crossover.

- Crossover operators preserving relative order:

  Modified crossover,

  Order crossover,

  Linear order crossover,

  Order-based crossover,

  Position-based crossover.

Please note that we also consider this classification for our experiments: We will analyze whether any of these groups performs generally better than the other.

6.2.1 Partially-Mapped Crossover

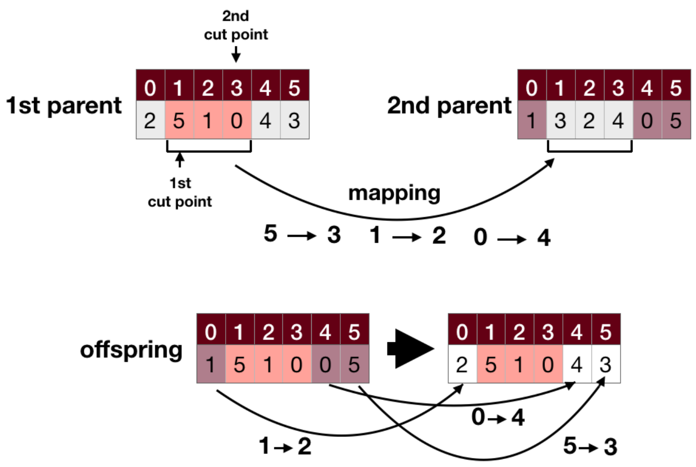


Figure 18: Partially-mapped crossover between parent chromosomes 251043 and 132405.

Partially-mapped crossover (**PMX**) as discussed by Potvin [21] takes two cut points. The cities which stand between these cut points in the first parent chromosome go directly to the offspring at exactly the same positions. The remaining positions are filled with the cities from the second parent chromosome. In order to handle possible duplicates, a mapping between two parents chromosomes for the region between the two cut points is introduced.

Consider for instance the example from Figure 18 with parent chromosomes 251043 and 132405. Let us assume that the two cut points are 1 and 3. This means that the substring 510 of the first parent is directly copied into the offspring. When filling in the remaining positions based on the second parent, we get the offspring chromosome 1 510 05. To eliminate the duplicates we make a mapping between the elements between the cut points in the parents chromosomes: 510 is mapped to 324, i.e., 5 is mapped to 3, 1 is mapped to 2, and 0 is mapped to 4. When applying this mapping, we replace 1 with 2 in the offspring at the position 1 and get 251005. As we see, the updated chromosome is not final as 0 is duplicated as well, therefore it will be replaced by 4 according to the mapping and we get 251045. Now we have to deal with the last 5 which can be replaced by 3. As a result we get the offspring 251043.

One can also think of this procedure as applying two mappings to the second parent: First, a mapping m is applied for replacing the positions between the cut points. In our example, m is defined as follows: $3 \to 5$, $2 \to 1$, $4 \to 0$. That is, the sequence 324 in the second parent is replaced by 510. Afterwards, the mapping m is inverted, giving us $m^{-1} : 5 \to 3$, $1 \to 2$, $0 \to 4$. This inverse mapping is then applied to all remaining positions.

Multiple replacements at the same position can occur if both parents have the same city in the part of the tour between the chosen cut points. Consider for instance the example from Figure 19 with the first parent chromosome 051243 and the second parent 125430. Let the two cut points be 1 and 3. In this case, 5 is mapped to 2, 1 is mapped to 5 and 2 is mapped to 4. Again the part between the chosen cut points from the first parent (512) goes directly to the offspring at the same positions, whereas the rest is taken from the second parent. As a result, we get the offspring 1 512 30. The city 1 is duplicated in the offspring, so it will be replaced by another value according to the defined mapping, namely by 5. Now the offspring consists of 5 512 30. However, the city 5 is duplicated as well and is replaced by 2, resulting in the offspring 2 512 30. The city 2 is, however, duplicated as well, therefore another replacement will be made. The city 2 is replaced by 4 according to the mapping. As the cities 3 and 0 are not duplicated, no replacements are necessary. The resulting offspring is 4 512 30.

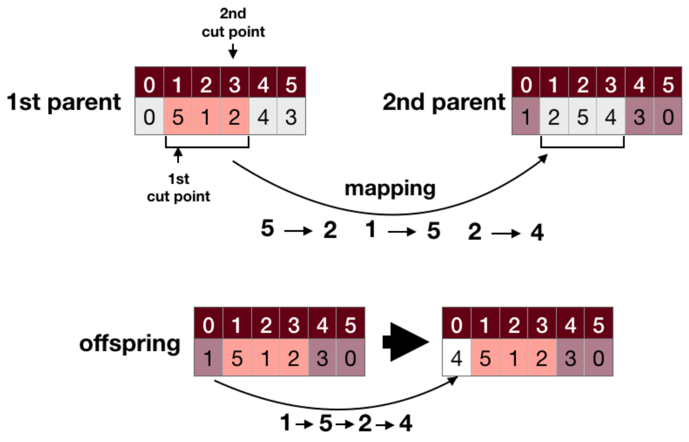


Figure 19: Partially-mapped crossover with multiple replacements between parent chromosomes 051243 and 125430.

6.2.2 Cycle Crossover

Cycle crossover (**CX**) as discussed by Potvin [21] looks for a cycle in the two parent chromosomes. A cycle consists of a set of cities which are found in the same set of positions in both parent chromosomes. For instance, let the first parent chromosome have the genes 0543162 and the second parent chromosomes have the genes 0435216. The set of cities $\{3, 4, 5\}$ is found in the same set of positions in both parents (namely $\{1, 2, 3\}$), constituting a cycle 5-4-3-5 (see Figure 20).

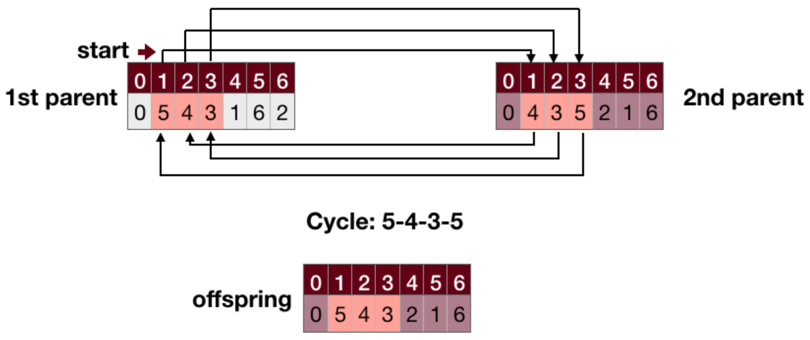


Figure 20: Cycle crossover between parent chromosomes 0543162 and 0435216.

To find a cycle, we start from the index where the parent chromosomes differ from each other. In our example, this means that we ignore the city 0 which occupies the same position 0 in both parents, as it will not make any changes to the offspring. So we start with the index 1, where the city 5 is found in the first parent. Then we look for a city which stands at the same index in the second parent chromosome. In our case it is the city 4. Then we look for the index in the first parent chromosome where this city stands. It is index 2. We find the city in the second parent chromosome which stands at this index as well, namely the city 3. Now we go to the first parent chromosome and see that the city 3 stands there at the index 3. In the second parent, the city 5 occupies the position 3 and we found a cycle 5-4-3-5.

The cities of this cycle are copied from the first parent chromosome to the offspring and occupy the same positions there. All the remaining positions are filled with the cities from the second parent chromosome at exactly the same positions. As a result, the position of each city in the offspring is inherited from one of the parents and the resulting offspring is 0543216. Therefore, each element comes from one parent together with its position.

Alternatively, to find a cycle we could start at a random index and not at the beginning. Moreover, instead of just filling the remaining positions of the offspring with the cities from the second parent, we could search for additional cycles and use them to populate the offspring. Consider the example from Figure 21 with the parent chromosomes 051463278 and 042315687. The first element is the same in both parents, so the city 0 goes directly to the offspring to the position 0. Here we have three cycles, namely 5-4-3-5, 1-2-6-1, 7-8-7. So, at first, the subset of cities $\{3, 4, 5\}$,which corresponds to the first cycle, goes to the offspring at the positions from the first parent. Then the subset of cities $\{1, 2, 6\}$, which corresponds to the second cycle, goes to the offspring at the positions from the second parent. Finally, the subset $\{7, 8\}$ goes to the offspring at the positions from the first parent again, resulting in the offspring 052413678.

6.2.3 Modified Crossover

Modified crossover (**MX**) as introduced by Davis [8] randomly chooses an index, at which the first parent chromosome will be cut into half. The cities before the cut point are taken from the first parent and go to the offspring at the same positions. The other

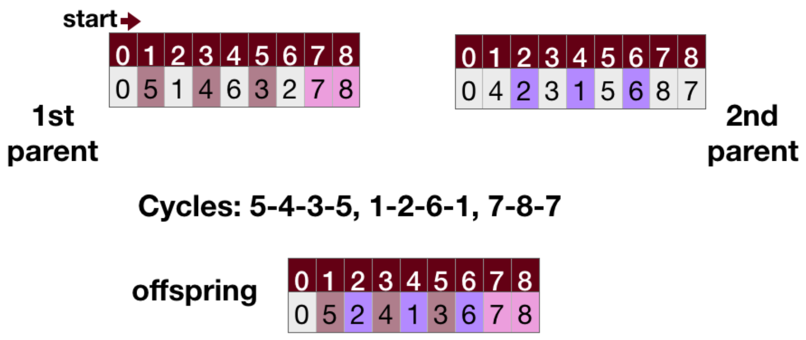


Figure 21: Cycle crossover while using multiple cycles between parent chromosomes 051463278 and 042315687.

indices of the offspring are filled with the cities from the second parent, starting at the beginning of the second parent chromosome. In order to avoid duplicates, the next city goes from the second parent to the offspring, only if this city has not been taken from the first parent chromosome yet. Otherwise, this city is ignored and the next city from the second parent goes to the offspring.

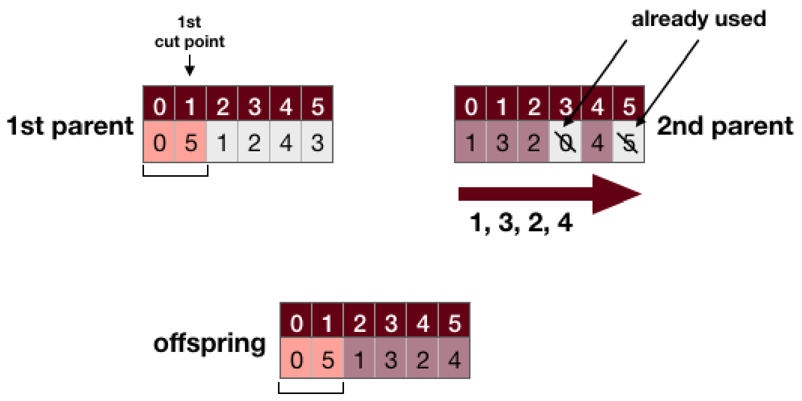


Figure 22: Modified crossover between parent chromosomes 051243 and 132405.

For instance consider Figure 22, where the first parent chromosome is 051243 and the second parent chromosome is 132045. Let the cut point be the index 1. Then, the cities which stand in the first parent before this cut point, namely 0 and 5, go directly to the offspring at the same positions. Now the cities from the second parent starting from its beginning are used: The city 1 goes to the offspring at the next free index, namely the index 2. So do the cities 3 and 2. The next city 0 has been already taken from the first parent chromosome, therefore it will be not used and the next city is taken into consideration, namely the city 4, which goes to the offspring at the last index. The last city 5 from the second parent has been already taken from the first parent, therefore we ignore it. The resulting offspring has the genes 051324.

6.2.4 Order Crossover

Order crossover (**OX**) as discussed by Potvin [21] takes two indices which correspond to two cut points. The part of the first parent chromosome between these indices is copied to the offspring directly at exactly the same positions. The other positions will be filled with the cities from the second parent chromosome in the following way: We start searching in the second parent at the index which stands just after the second cut point. If the observed city has already been used, when the cities were copied from the first parent, it will be ignored and the next city is considered. If the end of the second parent chromosome is reached during the search, the search continues at the beginning of it and continues until the second cut point is reached.

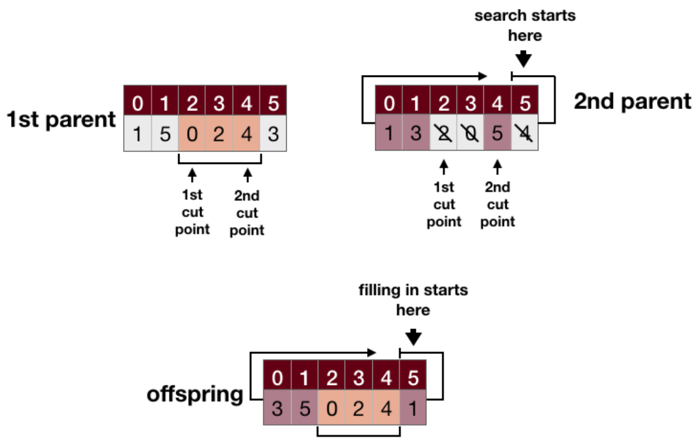


Figure 23: Order crossover between parent chromosomes 150243 and 132054.

For instance, consider the parent chromosomes 150243 and 132054 (see Figure 23). The chosen cut points are the indices 2 and 4. This means that the part of the first parent chromosome 024 goes directly to the offspring at the positions 2,3 and 4, respectively. Now the remaining part of the offspring will be filled. We start with the index after the second cut point and find the city 4 in the second parent chromosome. As this value has already been taken from the first parent chromosome, it cannot be used and the next city to be considered is the city 1 which stands at the index 0 in the second parent. This city has not been used yet, therefore it takes the position right after the second cut point, i.e. the position with the index 5, in the offspring. Now the end part of the offspring is full, therefore the next insertions continue at the beginning of the offspring chromosome, starting at the index 0. The next city in the second parent which is considered is the city 3. As it has not been used yet, it goes to the offspring at the position 0. The city 2 was already used, and so was the city 0. Then the city 5 is inserted at the position 1 in the offspring. Now, the second cut point in the parent chromosome is reached, and the offspring is filled. Therefore, the crossover is over, and the resulting offspring has the genes 350241.

6.2.5 Linear Order Crossover

Linear order crossover (**LOX**) was used by Akay and Yao [1] for the job shop scheduling problem. However, as this crossover operator works similar to the order crossover, which was discussed above, we will apply it to the TSP. The difference is that the remaining positions of the offspring are filled starting from the beginning of the offspring chromosome. Also the search of the next candidate city in the second parent starts at the beginning of the second parent chromosome and continues sequentially until its end is reached.

We consider the example which was used above for order crossover with the first parent chromosome 150243, the second parent chromosome 132054 and the indices 2 and 4 (see Figure 24). The offspring gets the genes 024 from the first parent at the positions 2,3 and 4, respectively. The remaining part of the offspring will be filled, starting with the first index 0. The candidate city for this position is searched in the second parent at the position 0, where the city 1 is found. As it has not been used in the offspring yet, it occupies the position 0 in the offspring. In the same way, the next city 3 from the second parent chromosome goes to the position 1 in the offspring. The positions 2 to 4 are already filled, therefore the next candidate from the second parent will go at the last index 5. And it will be the city 5, which can be taken into consideration in the second parent. Therefore, the resulting offspring has the genes 130245. Note that this operator works exactly as the order crossover does, if the cut interval is chosen at the end of the chromosome.

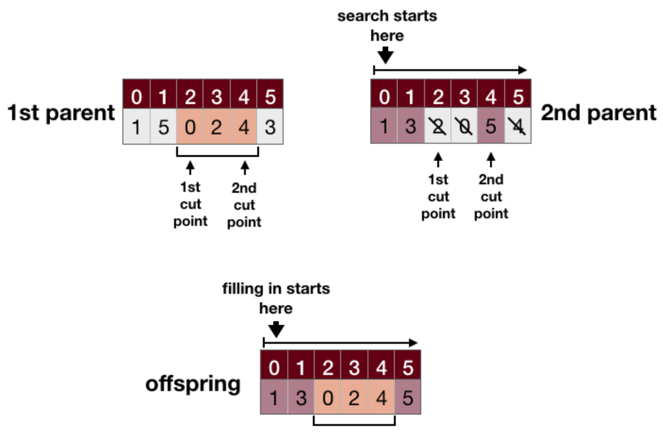


Figure 24: Linear order crossover between parent chromosomes 150243 and 132054.

6.2.6 Order-Based Crossover

Order-based crossover (**OBX**) as discussed by Potvin [21] randomly selects a subset of cities in the first parent chromosome and puts them into the offspring in exactly the same order in which these cities occur in the first parent, but at the positions, which these cities occupy in the second parent. Each remaining position in the offspring is filled with a city which occupies this position in the second parent. As a result, if the initial, randomly chosen subset of cities is very small, then the offspring is very similar to the second parent chromosome.

For instance, let the first parent chromosome be 051243 and the second parent chromosome be 132405 (see Figure 25). The chosen subset of cities is $\{3, 4, 5\}$. These cities appear in the first parent chromosome in the order 5, 4, 3. This order must be preserved. In the second parent, they occupy the positions 1, 3, 5. Therefore, these cities appear in the offspring in the defined order at the positions 1, 3, 5. The remaining positions are filled with the other cities from the second parent, preserving their positions. The resulting offspring has the genes 152403.

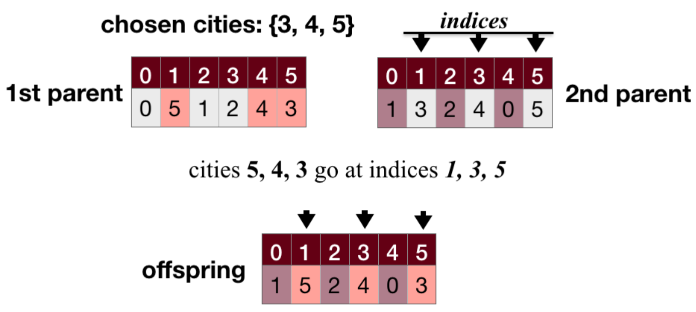


Figure 25: Order-based crossover between parent chromosomes 051243 and 132405.

6.2.7 Position-Based Crossover

Position-based crossover (**PBX**) as discussed by Potvin [21] takes randomly a subset of positions. The cities which occupy these positions in the first parent chromosome go directly to the offspring chromosome at the same positions. All the remaining positions of the offspring are filled with the cities from the second parent chromosome in the following way: Starting from the beginning of the second parent chromosome, the next city comes into consideration. If it was already taken from the first parent, then the city is ignored and the next city on the consecutive position comes into question. This operator therefore preserves the absolute order of the cities which are taken from the first parent chromosome and the relative order of the cities which are taken from the second parent chromosome.

For instance, take the parent chromosomes 150243 and 132054 (see Figure 26). The set of chosen positions is $\{1, 3, 4\}$. Cities 5, 2 and 4 occupy these positions in the first parent and are inserted into the offspring at these positions. Now the genes of the second parent chromosome are used. The city which occupies the starting position 0 in the second parent is the city 1. It has not been used yet, therefore, the city 1 goes to the offspring at the position 0. The next city in question is the city 3. It has not been used yet as well, therefore it occupies the next free position in the offspring, namely the position 2. The next two positions in the offspring are already filled. Therefore, the last position is filled with the city 0 from the second parent, which is the last city which has not been used yet. So, the resulting offspring has the genes 153240.

Note that this crossover operator is similar to linear order crossover. The only difference is that the PBX takes a subset of positions instead of an interval in the first parent chromosome.

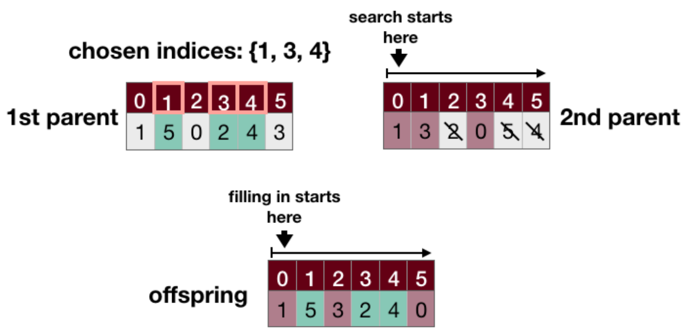


Figure 26: Position-based crossover between parent chromosomes 150243 and 132054.

6.3 Adjacency Representation Crossovers

The main aim of the crossover operators using this type of tour representation is to preserve as many edges from the parent chromosomes as possible, as this information is relevant for the TSP. Together with the ERX (see Section 6.4), these crossover operators are therefore called edge-preserving operators (see [21]). In each step, they use either an edge from the first parent chromosome or an edge from the second one. An exception is the last edge, which is normally not inherited from the parents but is added to the final solution in such a way that the Hamilton cycle is closed.

Please note that according to Potvin [21], "the edge-preserving operators are superior to the other types of crossover operators".

6.3.1 Alternate Edges Crossover

The alternate edges crossover (**AEX**) was described by Grefenstette et al. [11]. It alternately chooses the edges from the two parents. It includes the following steps:

1. Choose randomly an edge (s, u) from one of the parents as a starting point for the child tour.
2. Choose from the other parent an edge (u, v) which is incident to the chosen edge (s, u) and extend the tour.
3. If the chosen edge introduces a cycle, choose a random edge which does not introduce a cycle and extend the tour.
4. Extend the tour further by choosing the edges alternately from the parents until all the cities are included into the tour.

For instance, we have the following parent tours: 123450 and 140532. Let us assume that the edge $(0, 1)$ was randomly chosen from the first parent. It will be directly copied into the child (see Figure 27). The next edge $(1, 4)$, which is incident to the city 1, will be taken from the second parent and put into the child tour. The next edge $(4, 5)$ comes from the first parent. The next two edges are $(5, 2)$ and $(2, 3)$. They extend the tour, as it is shown in Figure 28. The last edge is supposed to be taken from the second parent and it would be the edge $(3, 5)$. However, it introduces a cycle and cannot be taken (see Figure 29).

Instead, the last possible edge $(3, 0)$ is chosen and finishes the tour. As this edge was the only possible edge to extend the tour, a random choice could not be applied in this case. The resulting offspring is 143052. According to Grefenstette et al. [11], the results of using this crossover were disappointing. Nevertheless, we include it as well in order to compare it to other crossovers.

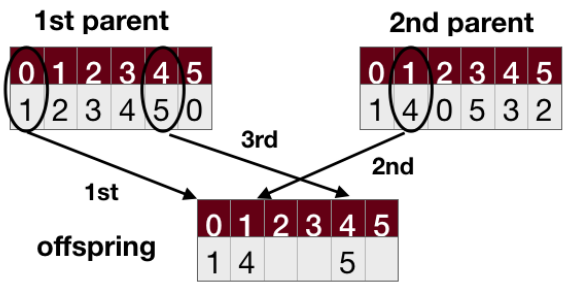


Figure 27: Alternate edges crossover: Introducing the edges (0, 1), (1, 4), (4, 5).

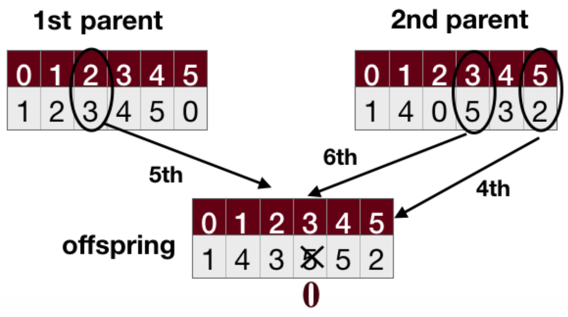


Figure 28: Alternate edges crossover: Introducing the edges (5, 2), (2, 3).

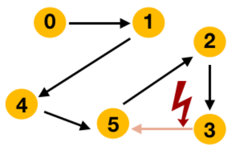


Figure 29: Alternate edges crossover: Cycle in the tour after adding the edge $(3, 5)$.

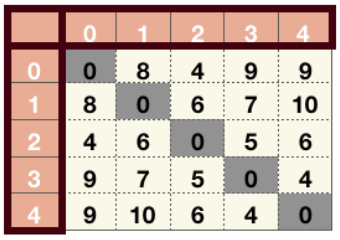
6.3.2 Heuristic Crossover

The heuristic crossover (**HX**) was introduced by Grefenstette et al. in [11] for improving the effectiveness of the genetic algorithm for the TSP. In general, genetic algorithms are free from domain information, which makes them widely applicable. For better performance on a specific domain, a so-called "hybrid" genetic algorithm can be introduced, where some domain-dependent information is used. The heuristic crossover considers the distances on the edges and looks for the shortest parental edge to a city, which has not been visited yet. The HX algorithm includes the following steps:

1. Choose randomly a city s as a starting point for the child tour.
2. Find an edge (s, u) in the first parent and an edge (s, v) in the second parent extending the tour from the current city s.
3. Choose the edge with a smaller distance. If this edge leads to a cycle in the current partial tour, choose randomly an edge which does not introduce a cycle.
4. Go to step 2 and continue to extend the current partial tour by choosing the shorter edge in the parents which extends the tour, until all the cities are included into the tour.

There exist some modifications to this algorithm. As discussed by Potvin [21], both Liepins et al. [19] and Suh and Van Gucht [28] proposed a modification of step 4: If the shortest parental edge leads to a cycle in the current partial tour, then the edge from the other parent will be used. If it also introduces a cycle, then extend the tour with a random edge that does not introduce a cycle.

Let us again make an example. Consider the parent chromosomes 13420 and 14302 (see Figure 31). Let us assume that the first edge was chosen at the index 0. As the 0th edge in the both parents is the same, the offspring gets this edge, i.e. 1 goes at the index 0. Now 1 is the next index. The edge $(1, 3)$ in the first parent has the distance 7, whereas the edge (1, 4) in the second parent has the distance 10 (see the distance table in Figure 30).



	0	1	2	3	4
0	0	8	4	9	9
1	8	0	6	7	10
2	4	6	0	5	6
3	9	7	5	0	4
4	9	10	6	4	0

Figure 30: Distance table for the HX example in Figure 31.

Based on its smaller distance, we choose the edge from the first parent and check whether it leads to a subcycle. The current tour of the offspring with this edge is 013. We do not have a subcycle here; therefore, the edge $(1, 3)$ goes to the offspring. The next index is 3. The edge $(3, 2)$ in the first parent with distance 5 wins against the edge $(3, 0)$ in the second parent having a distance of 9. The current offspring is 13-2- which corresponds to the tour $\pi = (0, 1, 3, 2)$. The next index 2 brings us to the edge $(2, 4)$ in the first parent with distance 6 and to the edge $(2, 3)$ in the second parent with distance 5. The edge $(2, 3)$ wins but it introduces a subcycle 3-2-3. For this reason, the other edge $(2, 4)$ is taken and the current offspring with this edge is 1342-. There is no choice for the last edge because we need a Hamilton cycle. Thus, getting the last edge $(4, 0)$, the resulting offspring is 13420 which corresponds to the tour $\pi = (0, 1, 3, 2, 4)$. Please note that it is identical to the first parent.

It is interesting to note that this crossover tends to repeat one of the parents, if the number of genes is small.

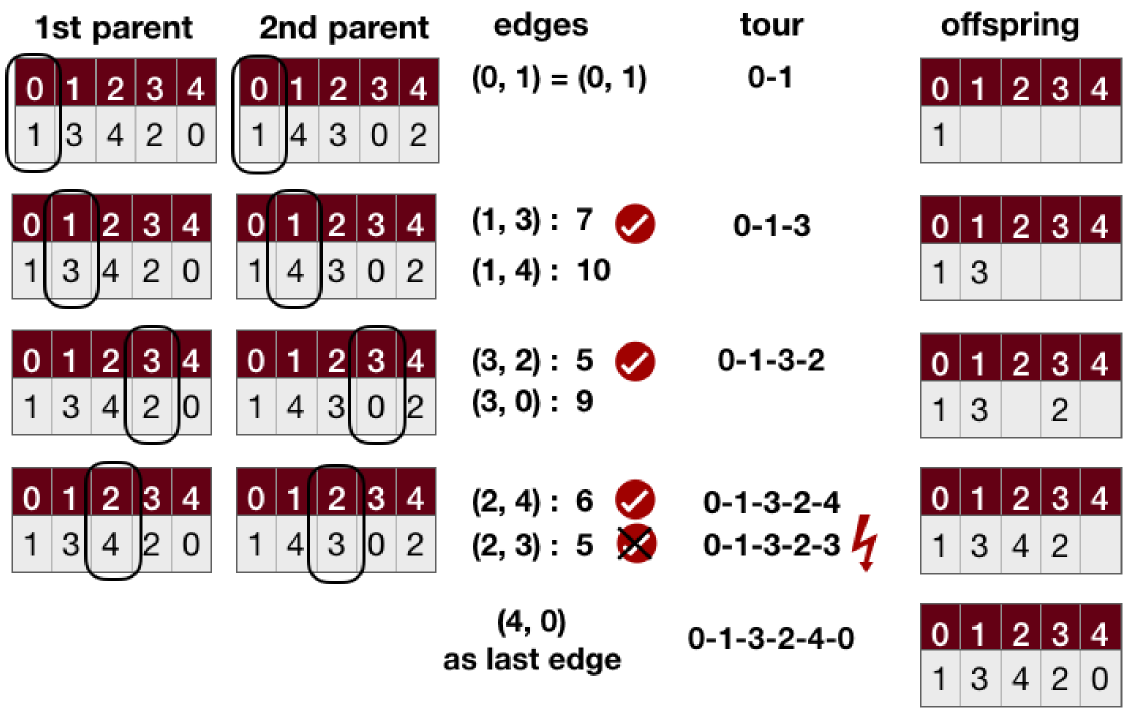
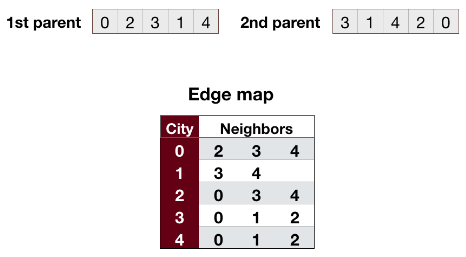


Figure 31: Heuristic crossover for parent chromosomes 13420 and 14302.

6.4 Edge Recombination Crossover

We consider the edge recombination crossover (**ERX**) developed by Whitley, Starkweather, and Fuquay [29] separately because it combines the features of the previous two groups. On the one hand, it works with the path representation like the path representation crossovers do. On the other hand, its aim is to transfer as many parental edges to the offspring as possible like the adjacency representation crossovers do. For this reason, it uses a special data structure called 'edge map'. This edge map contains for each city a list of cities, which are the neighbors of this city either in the first parent or in the second parent chromosome. That is, it lists for each city all adjacent cities in both parents.

For instance, assume that we have two parent chromosomes 02314 and 31420 in the path representation (see Figure 32). The city 0 has the adjacent cities 2, 3, 4 in both parents. The list for the city 1 includes 3 and 4. The city 2 has the neighbors 0, 3, 4 and so on.


1st parent
0 2 3 1 4
2nd parent
3 1 4 2 0
Edge map

City	Neighbors		
0	2	3	4
1	3	4	
2	0	3	4
3	0	1	2
4	0	1	2

Figure 32: Initial state of the edge map for the parent chromosomes 02314 and 31420.

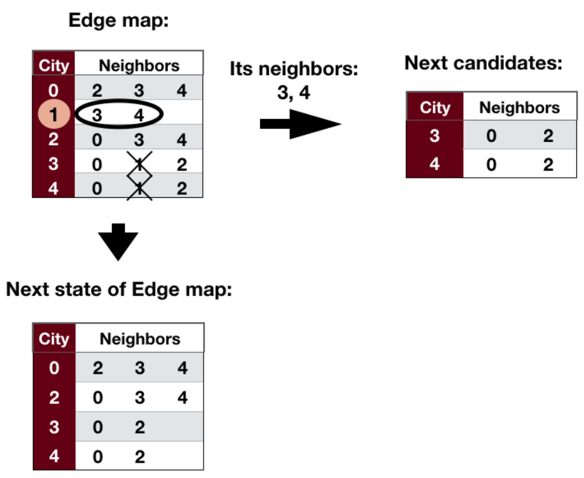


Figure 33: State of the edge map for the parent chromosomes 02314 and 31420 after choosing the city 1.

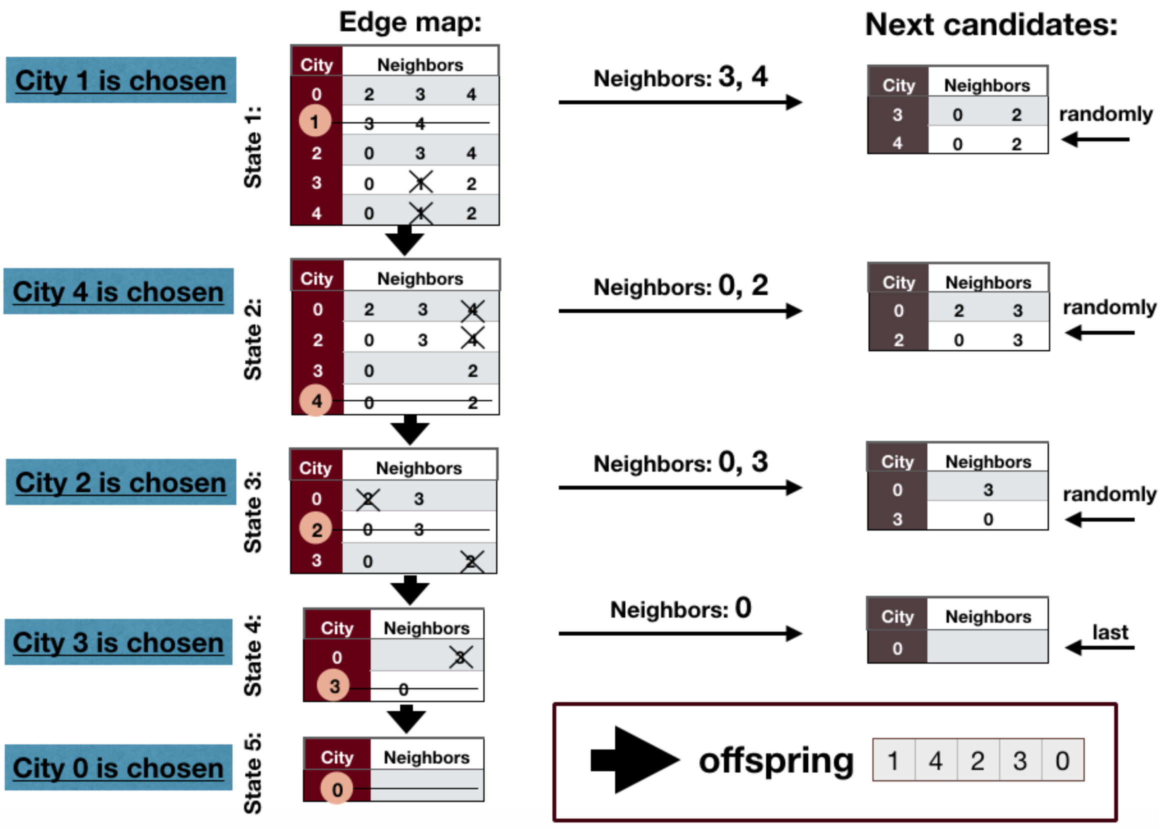


Figure 34: Edge recombination crossover for the parent chromosomes 02314 and 31420.

This is the initial state of the edge map because no cities have been included in the offspring yet. All the edges in the edge map are called "active", as they are available to continue the tour. Among these, ERX chooses the edge which leads to a city with the minimal number of active edges. In other words, a city with the smallest list of neighbors will be chosen. If there is more than one candidate city, the choice is made randomly among them. The edge map will be updated after each city selection, so that only active edges stay there.

We will demonstrate how the edge recombination crossover works using the above mentioned parent chromosomes and the constructed edge map from Figure 32.

The city with the shortest list of neighbors is the city 1. So, it will be selected and added to the tour. This city has neighbors 3 and 4. Therefore, these cities will be now considered as the next candidates to continue the tour and the city 1 will be deleted from the map (see Figure 33). Figure 34 shows how the edge map transforms while doing this crossover. So, the next step is to compare the candidates according to the length of their list of neighbors. The candidate with the shortest list will be chosen. If some candidates have lists of equal minimal length, as in our example the cities 3 and 4, then one of them will be taken randomly. Let us assume that the city 4 was randomly chosen. It will be added to the tour and deleted from the edge map. As this city has the cities 0 and 2 as neighbors, they are the next candidates to continue the tour. Both cities have the same number of neighbors, therefore one of them will be chosen randomly. Let us assume that the city 2 was chosen and is added to the tour. Its neighbors have the same number of neighbors as well; therefore, the city 3 is assumed to be chosen randomly. This city continues the tour and its only neighbor, the city 0, finishes the tour. The resulting offspring is 14230.

7 Mutation Operators

Mutation is one of the most important operators of genetic algorithms. It prevents the algorithm from getting stuck in a local minimum. It adds randomness in the evolutionary process and can help to find genes, which were lost and cannot be reproduced otherwise. In other words, mutation diversifies the population.

Mutation operators for the TSP make random permutations of the cities and they can therefore modify the tour to a large degree. The mutation operators which are discussed below will be applied only to the path representation. Applying them on the adjacency representation can ruin the tour. For instance, let us consider the following tour in the path representation 01432 which corresponds to the tour 14023 in the adjacency representation. If we swap two cities 1 and 3, it results in the tour 03412 in the path representation. The Hamilton cycle is still there. If we also swap the entries 1 and 3 in the adjacency representation, we get 34021. The corresponding graph in Figure 35 illustrates that there exist two cycles and the Hamilton cycle was destroyed.

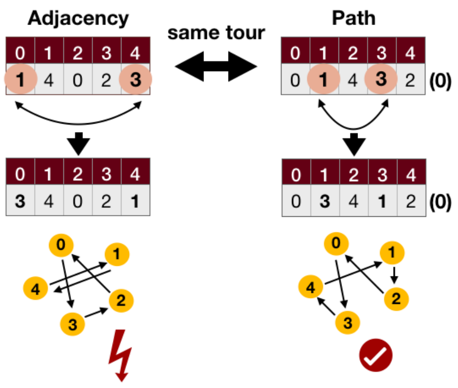


Figure 35: Swapping cities in adjacency and path representation.

The **Swap mutation** operator as discussed by Potvin [21] modifies the tour slightly, only exchanging the positions between two cities. Given the chromosome 014532 the swap mutation between the cities 1 and 3 which occupy the positions 1 and 4, respectively, transforms the chromosome into 034512 (see Figure 36).

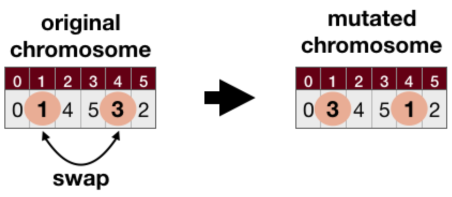


Figure 36: Swap mutation for the chromosome 014532 when swapping the cities 1 and 3.

The **Scramble mutation** operator as discussed by Potvin [21] takes two indices, which define two cut points. The cities between these cut points are randomly permuted.

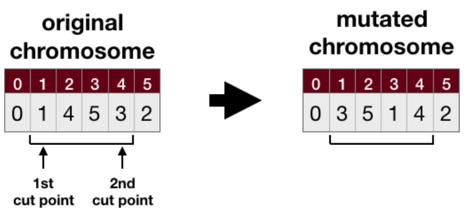


Figure 37: Scramble mutation for the chromosome 014532 with the cut points 1 and 4.

Given the chromosome 014532, the scramble mutation for the cut points 1 and 4 can generate, for instance, the chromosome 035142 (see Figure 37).

The **Shift mutation** operator as used by Akay and Yao [1] has two random components: a random city and a random integer. The chosen city will be shifted to the left or to the right, depending which shift is used. The chosen integer corresponds to the number of positions by which the chosen city will be shifted. Given the chromosome 014532, let us assume that the random city is the city 1 and the random integer is 3. We will use the right shift. Therefore, the city 1 will be shifted to the right by 3 positions (see Figure 38).

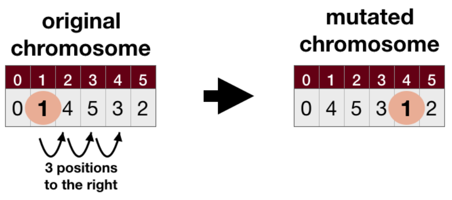


Figure 38: Right shift mutation for the chromosome 014532 with random city 1 and random integer 3.

If the chosen integer is greater than the number of remaining steps to the right (or to the left, if the left shift is chosen), then the shift continues at the beginning of the chromosome. For instance, if the random integer was 5 in the example above, then the city 1 will be at the position 0 (see Figure 39).

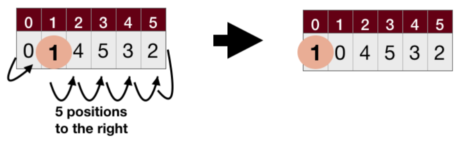


Figure 39: Right shift mutation for the chromosome 014532 with random city 1 and random integer 5.

The **Inversion mutation** operator as used by Akay and Yao [1] takes two indices which define two cut points. The order of the cities between them is inverted. Given the chromosome 014532, the inversion mutation for the cut points 1 and 4 generates the chromosome 035412 (see Figure 40).

The **Insertion mutation** operator as used by Akay and Yao [1] takes two indices, which define two cut points. The cities which occupy the positions between the first

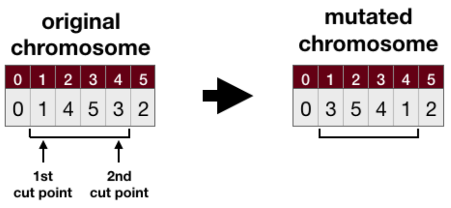


Figure 40: Inversion mutation for the chromosome 014532 and cut points 1 and 4.

cut point (exclusively) and the second cut point (inclusively) are shifted one position to the left. The city which occupied the index, which corresponds to the first cut point, now goes at the index, which corresponds to the second cut point, i.e. this stands after the set of shifted cities now. Given the chromosome 014532, the insertion mutation for the cut points 1 and 4 generates the chromosome 045312 (see Figure 41).

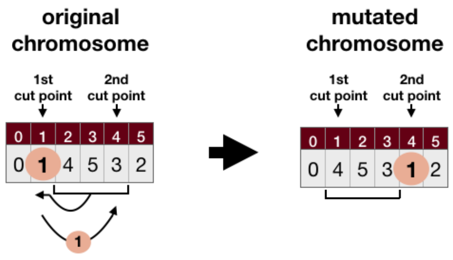


Figure 41: Insertion mutation for the chromosome 014532 and cut points 1 and 4.

The **Displacement mutation** operator [1] takes three indices, which define three cut points. The cities which occupy the positions between the first cut point (inclusively) and the second cut point (inclusively) are shifted at the position after the third cut point. This means that the cities between the second cut point (exclusively) and the third cut point (inclusively) are moved to the left, where the first cut point was. Given the chromosome 01453276 the displacement mutation for the cut points 1, 4 and 6 generates the chromosome 02714536 (see Figure 42).

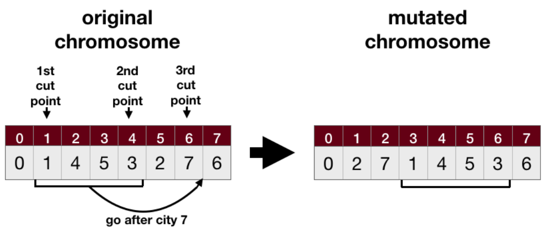


Figure 42: Displacement mutation procedure for the chromosome 01453276 and cut points 1, 4, and 6.

8 Tuning General Parameters

The main research topic of this thesis is to find out which variation of genetic algorithms works best in solving the TSP. In order to provide an answer, this chapter considers first how the initial population can be constructed and which settings for the general parameters should be used.

8.1 Preliminaries

We used the symmetric instances of the TSPLIB library (see [17]) for our experiments. Since this dataset uses a variety of different file formats, it was necessary to make some preprocessing steps and create a unified format to work with the data. We assume that the number of cities can influence the performance of our genetic algorithm; therefore, we have divided the data into five groups of similar size:

- TSP instances with less than 100 cities (22),
- TSP instances with 100 to 200 cities (24),
- TSP instances with 200 to 500 cities (18),
- TSP instances with 500 to 1500 cities (23), and
- TSP instances with 200 to 500 cities (15).

Some of the instances have a known optimal solution which can serve as a benchmark in the experiments. Let f_i be the fitness value of the chromosome returned by the genetic algorithm for a TSP instance i and f_i^{opt} be the fitness value of the chromosome which encodes an optimal tour for i. In order to measure the effectiveness of different operators in our genetic algorithm, we introduce the index:

$$E_{opt}(i) = (f_i^{opt} - f_i)/f_i^{opt}$$

This index corresponds to the relative error of the fitness value f_i with respect to the optimal value f_i^{opt}. Moreover, we denote by E_{opt}^t the average of $E_{opt}(i)$ over t instances. A most effective configuration of the genetic algorithm is one which minimizes E_{opt}^t.

Most of the TSP instances come however without an optimal solution. Moreover, the used library (see [17]) does not provide any information about the best known solutions for these instances as well. Therefore, for these instances, the results of our experiments will be compared to the solutions provided by construction heuristics. Therefore, we introduce the index:

$$E_h(i) = (f_i^h - f_i)/f_i^h$$

f_i^h is the fitness value of the chromosome which encodes a shortest tour delivered by construction heuristics within ten minutes. This index corresponds to the relative error of the fitness value f_i with respect to the fitness value f_i^h. Moreover, let E_h^t be the average of $E_h(i)$ over t instances. A most effective configuration of the genetic algorithm is one which minimizes E_h^t. For the cases where the genetic algorithm shows better results than the heuristics, we will use another index:

$$I_h(i) = (f_i - f_i^h)/f_i^h$$

This index corresponds to the improvement made by the genetic algorithm over f_i^h. I_h^t is an averaged $I_h(i)$ over t instances. We want to maximize I_h^t.

Please note that all the following experiments have been conducted using a Genuine Intel CPU @1601.320MHz with 9 GB memory. In each experiment, the genetic algorithm has a time limit of ten minutes. The program code and all the raw data are available in [4].

As already mentioned in Section 2.2, there are two main ways to construct an initial population, namely, randomly or using heuristics. We are going to use both variants in our experiments. We have implemented the following construction heuristics: nearest neighbor (see Section 4.1), double nearest neighbor (see Section 4.2), nearest insertion (see Section 4.3), and farthest insertion (see Section 4.4). All of these heuristics are deterministic, but their results may depend on the start city. In order to initialize the population, we repeatedly ran each heuristic with a new randomly drawn start city until a given time limit of ten minutes was reached. This was done for each TSP instance individually. In the best case, the number of different tours is equal to the size of the TSP instance. Naturally, some of the tours which will be produced by the heuristics represent the same tour. We only include the chromosomes containing different tours in the initial population.

Please note that the above mentioned heuristics were launched separately from the genetic algorithm. During ten minutes, we sequentially started one heuristic after another with a randomly drawn start city. When ten minutes ran out, we stored the results to use them in our genetic algorithm. Thus, the time needed for the random initialization of the population does not significantly differs from the time needed to initialize the population using the results of heuristics. If the number of different tours provided by the heuristics is smaller than the desired population size, the remainder of the population is filled with random permutations of the cities. We also ensure that these random chromosomes represent different tours.

In addition to this heuristic-based initial population, we will also consider an initial population which does not contain results of the heuristics, that is, which is exclusively based on random permutations.

8.2 Experimental Setup

In our first experimental step, we want to analyze which configuration of two general parameters, namely population size and mutation rate, results in the best performance. In order to do so, we will study how the fitness value changes when varying these parameters while keeping the type of crossover, selection and mutation fixed. We have chosen the order crossover because, according to the literature (e.g., see [27]), it has shown better results than the other path representation crossover operators. Moreover, the edge preserving operators (AEX, ERX, and HX) cannot find good solutions for larger TSP instances according to Potvin [21]. Among the mutation operators, we have chosen the shift mutation as it works on the whole chromosome and not only on a part of it. We therefore assume that it may result in larger changes. Finally, the tournament was chosen as selection operator, as it is easy to implement and gives good results (see Section 5.3).

We have considered two groups of TSP instances:

- TSP instances with less than 100 cities, and
- TSP instances with 200 to 500 cities.

The motivation for this choice is that the instances are still small enough so that we can provide quite a large number of iterations. Besides, we assume that best configurations of parameters in these two groups could differ. Therefore, the experiments will be made for each group of instances separately.

As it was highlighted by Zhang et al. [30] who analyzed the effects of population size on the performance of genetic algorithms, "we do not have any reliable theory for choosing N", where N is the population size. The choice of an adequate population size can be difficult. On the one hand, if the size of the population is set too small to decrease the costs of calculations, the genetic algorithm has a poor choice of chromosomes which it can use; therefore, it can be hard for the genetic algorithm to find chromosomes with a good fitness value. On the other hand, very large populations can cause very high processing time and very slow convergence.

The common practice for setting the size of the population consists of using fix numbers like 20, 100, 200, 300, and 500 (Zhang et al., [30]) or 1000, 5000, and 10000 (Rexhepi, Maxhuni, and Dika, [23]) without considering the size of the TSP instances used in experiments. As one can see, the numbers used by different researchers are quite different; therefore we would like to use another approach inspired by Chen, Montgomery, and Bolufé-Röhler [7]. They investigated the curse of dimensionality in population-based algorithms and suggested to take into consideration the size of an instance when defining the population size. They do so by introducing a population size factor. Multiplying this factor with the size of an instance results in the size of the population. Although their research was conducted on particle swarm optimization and differential evolution algorithms, this approach can be used for genetic algorithms as well.

For both groups of instances we consider, it is possible to set the size of the population larger than the tour length to provide diversity in the population. Setting it smaller or even equal to the number of cities did not show promising results for both groups in preliminary investigations. A single iteration for this group of instances takes very short time. As a result, the genetic algorithm can produce a great number of iterations and slow convergence is therefore not expected to be a problem here. Thus, we consider the following options for the size factors s which is multiplied with the number of cities to obtain the population size: $s_1 = 3$, $s_2 = 5$.

Let us now consider the second general parameter, namely the mutation rate. There are several studies trying to optimize this hyperparameter, including the study by Beed et al. [5] about defining the GA parameters for solving Multi-Objective TSP. They considered mutation rates of 1%, 3%, 5%, and 10%. A mutation rate of 1% did not show promising results on our instances in preliminary experiments and was excluded from our further consideration. We therefore use the following mutation rate values m in our analysis: $m_1 = 3\%$, $m_2 = 5\%$, and $m_3 = 10\%$.

Overall, we thus have $2 * 3 = 6$ possible combinations for mutation rate and size factor. However, before analyzing them, we have conducted one more small preliminary experiment with respect to the tournament selection. We used the instances of the first group (with less than 100 cities), took these 6 combinations of mutation rate and size factor and considered three variants for the number of participants p in the tournament: randomly chosen from a small interval [2, 10], fixed at the expected value of this interval, namely 6, and fixed at a smaller value 3.

After that, we conducted a grid search on parameter settings for size factor and mutation rate evaluating all 6 possible combinations. So we used the following setting for our genetic algorithm:

- order crossover,
- shift mutation with the mutation rate m, where $m \in \{3\%, 5\%, 10\%\}$,
- tournament selection with the number of participants to be defined in a preliminary experiment,
- population size factor s, where $s \in \{3, 5\}$, and
- a time limit of ten minutes.

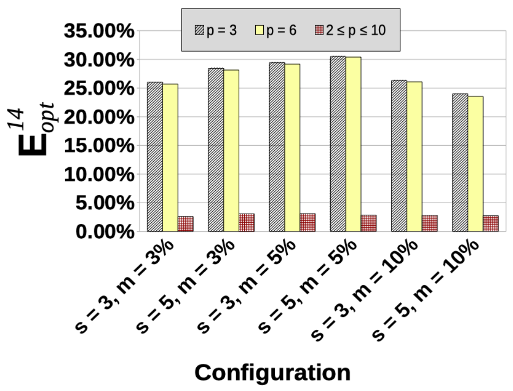
The experiments in this step were conducted without using heuristics in the population, that is, the population was initialized with random permutations. This choice was motivated by the results of several preliminary trials, where the heuristics were used: For some small instances with less than 100 cities the heuristics achieved the optimal values and for many instances of this group were close to the optimum; therefore, the results of different configurations for genetic algorithm showed similar results and could not be compared properly.

Please note that as genetic algorithms are nondeterministic, each configuration is always launched with 3 different seeds. The results are averaged over these three runs.

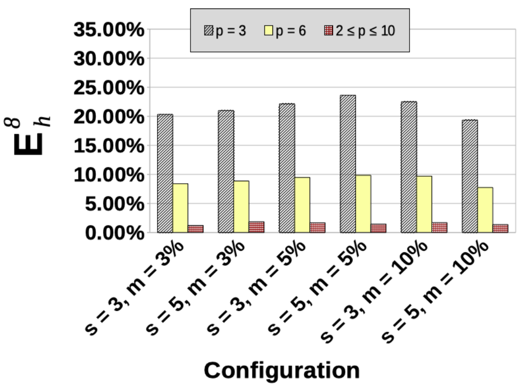
8.3 Results and Discussion

Concerning the number of participants for the tournament selection, the first option where we used a random number drawn from the interval [2, 10] showed the best results (see Figure 43); therefore, we will use this approach.

Now, we look for the best combination of parameters for two groups of instances, namely for the ones with less than 100 cities and the ones with 200 to 500 cities. For

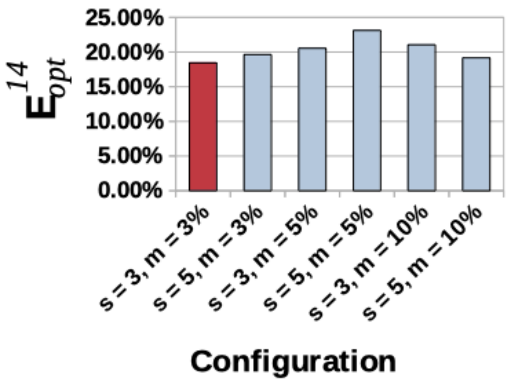


(a) Relative error with respect to the optimal value averaged over 14 instances.

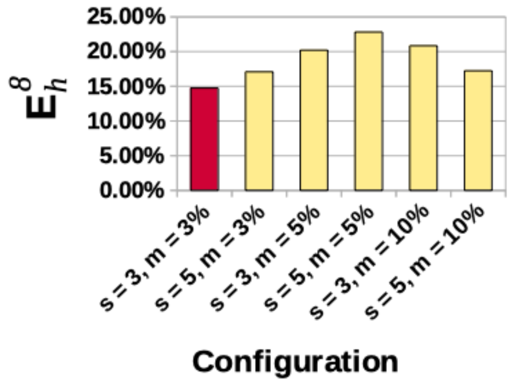


(b) Relative error with respect to the best results of heuristics (with ten minutes limit) averaged over 8 instances.

Figure 43: Relative error for 22 instances with less than 100 cities for 6 settings of general parameters and different variants of the tournament selection.



(a) Relative error with respect to the optimal value averaged over 14 instances.



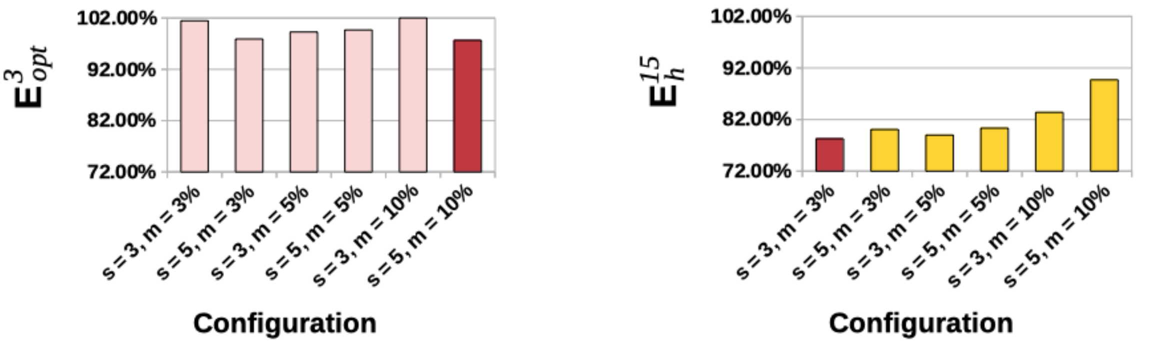
(b) Relative error with respect to the best results of heuristics (with ten minutes limit) averaged over 8 instances.

Figure 44: Relative error for 22 instances with less than 100 cities for 6 settings of general parameters.

the group of instances with less than 100 cities, the combination $s = 3, m = 3\%$ showed the best results for both indices E^{14}_{opt} and E^{8}_{h} (see Figure 44a and Figure 44b).

For the group of instances with 200 to 500 cities there are only 3 instances with a known optimal value. For them, the best configuration was $s = 5, m = 10\%$ (see Figure 45a). The other 15 instances have no known optimal value. Their results with respect to the best results of heuristics showed that the configuration $s = 3, m = 3\%$ was the best (see Figure 45b).

The differences among the configurations for the first group with less than 100 cities were much smaller as for larger instances of the second group with 200 to 500 cities. So, perhaps, the choice of such parameters as mutation rate and population size plays a more important role for larger instances.



(a) Relative error with respect to the optimal value averaged over 3 instances.

(b) Relative error with respect to the best results of heuristics (with ten minutes limit) averaged over 15 instances.

Figure 45: Relative error for 18 instances with 200 to 500 cities for 6 settings of general parameters.

The best configuration was different only for the case with 3 instances within the group with 200 to 500 cities, while for all other instances the same configuration, namely $s = 3, m = 3\%$, was the best. We consider the different result of these three instances as an outlier and choose the configuration $s = 3, m = 3\%$ to work with further because it worked at best for the most instances.

It is interesting to mention that while for smaller instances with less than 100 cities the error was smaller than 25%, for larger instances with 200 to 500 cities, the error exceeds 75%. This can be explained by the fact that the genetic algorithm completes less iterations in case of larger instances; therefore, it has not enough time to come to better solutions as it can do for smaller instances.

Mutation rate m:	*3 %*		*5 %*		*10 %*	
Size factor s:	E^{14}_{opt}	E^{8}_{h}	E^{14}_{opt}	E^{8}_{h}	E^{14}_{opt}	E^{8}_{h}
3	18,43 %	14,75 %	20,54 %	20,21 %	21,05 %	20,84 %
5	19,63 %	17,09 %	23,12 %	22,82 %	19,17 %	17,23 %

Table 4: Results for six combinations of general parameters for instances with less than 100 cities.

As there are significant differences in E_{opt} and E_h between two groups of instances, it makes sense to continue comparing the results among different size groups and not averaging over all instances.

Finally, if we take a look at Table 4, we can see that for small mutation rates like 3%, a smaller size factor is better. At the same time, for large mutation rates like 10%, a larger size factor performs better. This means that if there occur many mutations, the population has to be large, at least for small instances with less than 100 cities.

9 Comparison of Mutation and Crossover Operators

In this chapter, we will compare the effectiveness of different crossover types in combination with different mutation operators inside each of the group of crossover operators: path representation (in Section 9.1) and adjacency representation crossovers (in Section 9.2). Moreover, ordinal representation with OPX is discussed in Section 9.3. The crossover types which showed the best results inside each group combined with the most effective mutation operator will be considered again in the next phase.

According to the results from Section 8.3, we fix the mutation rate at 3% and the population size factor as 3 for the following experiments. Like in the previous section, we will use the same evaluation metrics and the same type of selection, namely tournament selection, where the number of participants in the tournament is drawn randomly from the interval $[2, 10]$. The experiments will also be made for each group of instances separately, as we assume that the size of the TSP instances can influence the results.

Please note that due to the shortage of computational time, we will conduct these experiments without using heuristics in the population, that is, the population is initialized with random permutations.

9.1 Path Representation Crossovers

9.1.1 Methods

In this part of our experiments, we will compare the effectiveness of path representation crossover operators with each other, namely modified crossover (MX), order-based (OBX), order crossover (OX), linear order crossover (LOX), position-based crossover (PBX), partially-mapped crossover (PMX), and cycle crossover (CX) (see Section 6.2). In addition, we combine these 7 crossover types with 6 different mutation types, namely swap, shift, scramble, inversion, insertion, and displacement (see Section 7). As a result, we get 42 configurations in total.

9.1.2 Results and Discussion

In the group of instances with less than 100 cities the OX showed good results in combination with all mutation types (see Figure 46). One reason for this may be the fact that we used the general parameters which were optimized for this crossover type. However, its performance is much better than the one of other crossover types, and its effectiveness is pointed out in the literature as well ([21]).

The OBX performed worst independent from the type of mutation used with a significant difference from the other crossover types. As it was described in Section 6.2.6, if the OBX receives a very small subset of cities to work with, the offspring will be very similar to the second parent chromosome. Perhaps, in our experiments the OBX

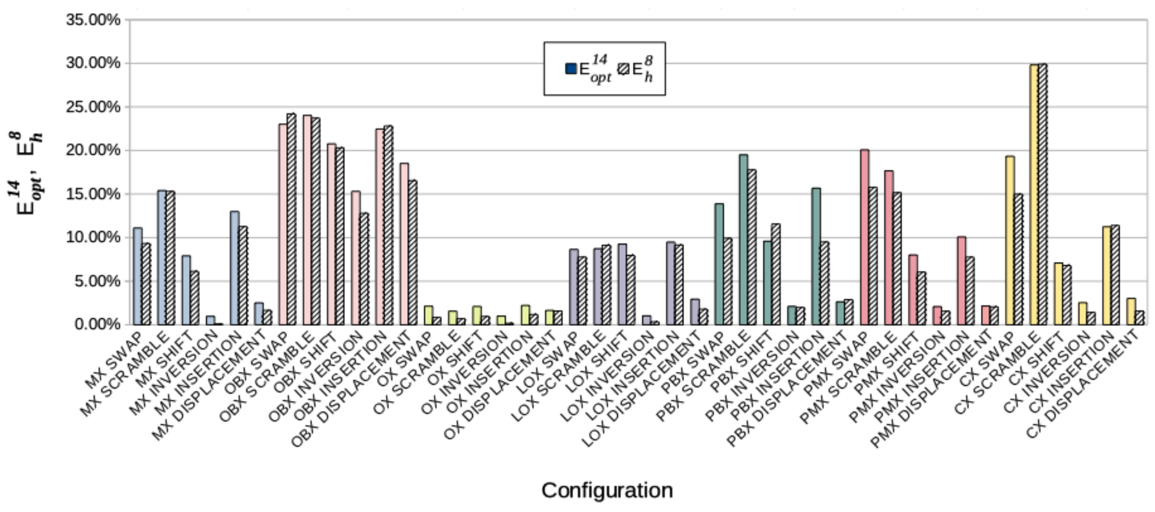


Figure 46: E^{14}_{opt}, E^{8}_{h} for instances with less than 100 cities for path representation crossovers.

	MX		OBX		OX		LOX		PBX		PMX		CX	
	E^{14}_{opt}	E^{8}_{h}	E^{14}_{opt}	E^{8}_{h}	E^{14}_{opt}	E^{8}_{h}	E^{14}_{opt}	E^{8}_{h}	E^{14}_{opt}	E^{8}_{h}	E^{14}_{opt}	E^{8}_{h}	E^{14}_{opt}	E^{8}_{h}
SWAP	11,10 %	9,33 %	23,01 %	24,20 %	2,13 %	0,85 %	8,64 %	7,77 %	13,88 %	9,93 %	20,07 %	15,77 %	19,34 %	14,99 %
SCRAMBLE	15,40 %	15,31 %	24,03 %	23,72 %	1,56 %	0,70 %	8,73 %	9,13 %	19,50 %	17,79 %	17,66 %	15,17 %	29,82 %	29,90 %
SHIFT	7,91 %	6,13 %	20,76 %	20,30 %	2,10 %	0,95 %	9,25 %	7,96 %	9,58 %	11,56 %	8,00 %	6,05 %	7,09 %	6,80 %
INVERSION	0,97 %	0,10 %	15,31 %	12,80 %	0,99 %	0,17 %	1,02 %	0,34 %	2,10 %	1,99 %	2,08 %	1,56 %	2,54 %	1,43 %
INSERTION	12,99 %	11,27 %	22,46 %	22,81 %	2,22 %	1,18 %	9,48 %	9,15 %	15,67 %	9,50 %	10,07 %	7,77 %	11,24 %	11,39 %
DISPLACE MENT	2,51 %	1,65 %	18,52 %	16,55 %	1,64 %	1,58 %	2,93 %	1,79 %	2,63 %	2,89 %	2,15 %	2,05 %	3,03 %	1,57 %

Table 5: Results for 22 instances with less than 100 cities for 7 path representation crossovers and 6 mutation operators

worked on average on too small subsets, and as we used random permutations to build the initial population in these experiments, the second parent was more probable to encode a relatively bad tour. Thus, the OBX may have often resulted in a weak offspring.

Concerning the mutation operators (see Table 5), the scramble mutation can be considered as less effective in general while the inversion and displacement mutation operators have shown good results. This can be caused by the scramble mutation making too many random permutations and thus destroying many edges while the inversion and displacement mutation preserve more edges from the parents (see Section 7).

If we take a look at the concrete combinations, the best results were shown by the

MX, the OX, and the LOX with inversion, where the MX with inversion mutation is considered as the winner configuration.

In the group of instances with 200 to 500 cities, the OX has shown good results in combination with all mutation types as well (see Figure 47). The OBX has shown worst results in combinations with all mutation operators.

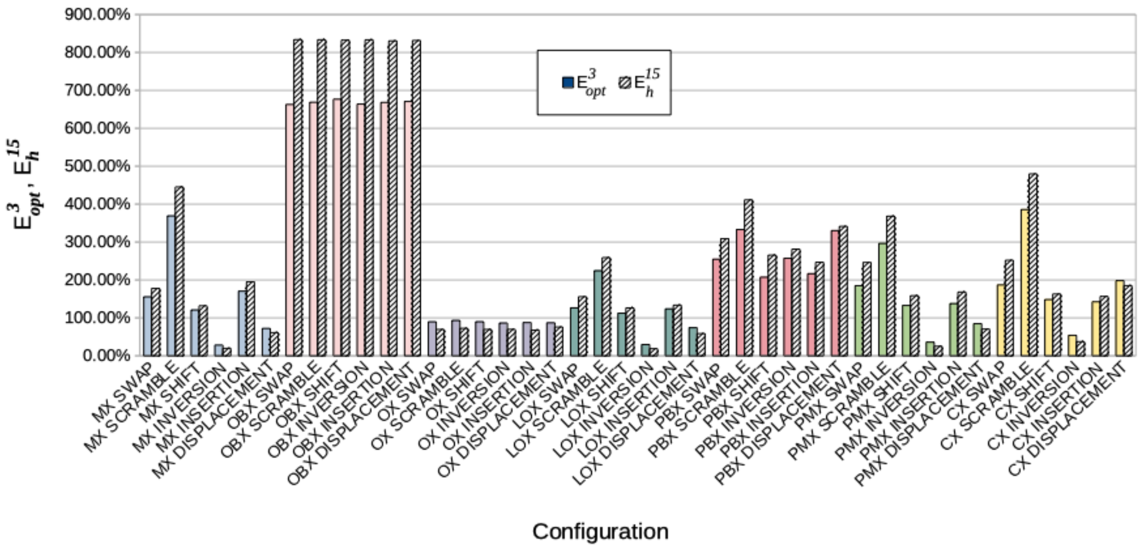


Figure 47: E^{14}_{opt}, E^{8}_{h} for instances with 200 to 500 cities for path representation crossovers.

	MX		OBX		OX		LOX		PBX		PMX		CX	
	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h
SWAP	155,44 %	177,76 %	662,84 %	833,80 %	89,56 %	69,35 %	126,36 %	155,82 %	254,67 %	308,82 %	184,91 %	246,21 %	187,00 %	251,94 %
SCRAMBLE	368,96 %	445,29 %	668,63 %	833,52 %	93,19 %	72,46 %	224,33 %	258,98 %	332,97 %	411,25 %	296,33 %	368,15 %	385,66 %	479,69 %
SHIFT	120,79 %	131,93 %	676,71 %	832,47 %	89,76 %	69,47 %	112,34 %	126,31 %	207,35 %	265,47 %	132,89 %	158,51 %	148,34 %	162,92 %
INVERSION	28,24 %	19,71 %	663,82 %	833,32 %	86,38 %	69,72 %	29,75 %	18,81 %	257,31 %	281,32 %	35,91 %	25,19 %	53,98 %	37,27 %
INSERTION	170,40 %	194,90 %	668,20 %	830,54 %	87,67 %	67,71 %	123,59 %	133,64 %	216,46 %	246,73 %	137,68 %	168,13 %	142,50 %	156,65 %
DISPLACE MENT	72,11 %	61,10 %	670,70 %	831,59 %	87,05 %	75,86 %	74,09 %	58,50 %	330,11 %	341,35 %	84,66 %	70,50 %	198,24 %	185,05 %

Table 6: Results for 18 instances with 200 to 500 cities for 7 path representation crossovers and 6 mutation operators

Among the separate configurations, the MX, the LOX, and the PMX all combined with the inversion mutation performed at best. The LOX with inversion mutation is considered as the winner configuration, as it is slightly better than the MX concerning

E_h^{15} category, where 15 instances were used in comparison with only 3 instances for E_{opt}^3.

If we take a look at the results grouping them according to mutation types, the inversion and displacement mutation operators seem to be generally better than the other types while the scramble mutation has showed relatively bad results (see Table 6).

In Section 6.2, we have considered two groups of crossover operators, namely crossover operators preserving absolute positions (PMX, CX) and crossover operators preserving relative order (MX, OBX, OX, LOX, PBX) to analyze whether one of the groups is generally better than the other. According to Potvin [21] the crossover operators which preserve relative order perform much better than the crossover operators which preserve absolute order. In his review he referred to the results made by Oliver, Smith, and Holland [20] and the results made by Starkweather et al. [27]. Oliver, Smith, and Holland compared only three operators, namely the OX, the PMX, and CX. Our results confirmed their studies: the OX has performed much better than the PMX and the CX. Of course, the OX has got an advantage because the hyperparameters were tuned for it, but still the difference in performance which it showed is considerable.

If we take a look at the experiments conducted by Starkweather et al. [27], they considered 5 crossover operators of the path representation type, namely the OX, the OBX, and the PBX which preserve relative order, and the PMX and the CX which preserve the absolute positions. They used only one very small TSP instance with 30 cities fixing population size at 500 with 50000 generations and no mutation was used with one exception: Some unspecified mutation was used in the CX if the offspring and one of the parents were identical. In their study, the group of crossover operators preserving the relative order has shown much better results than the group with the ones preserving the absolute positions. The OX has performed at best with a strong dominance. Our results were averaged over 40 instances, always used mutation and considered two more representatives of the group of crossovers preserving the relative order, namely MX and LOX. Our results confirmed the dominance of the OX. In difference to our experiments, the OBX has however shown not the worst results here. The PMX and the CX performed worst. However, these two last operators used for some unknown reason larger population of 1400 and 1500 chromosomes, correspondingly, while the other operators worked with 1000 chromosomes. If we do not consider the OBX which performed worst in our experiments, in general the operators of the group preserving the relative order performed better than the operators of the first group. It is interesting to mention that the MX was not used in the mentioned studies, although it has shown one of the best results in our experiments.

It is obvious that for larger instances with 200 to 500 cities the genetic algorithm performed approximately 10 times worse than for smaller instances with less than 100 cities. This could be perhaps explained by the fact that the genetic algorithm made less iterations for larger instances; therefore, it had no enough time to discover good solutions.

Finally, as it was mentioned in Section 6.2, the LOX works similar to the OX, and the PBX is similar to the LOX. It is interesting that in these experiments, the OX has performed better on average than the LOX, and the LOX has shown better results than the PBX. However, this tendency could be influenced by the choice of the used parameters which were tuned specifically for the OX. Nevertheless, due to its good performance here, we will consider the OX together with the MX and the LOX further in our experiments.

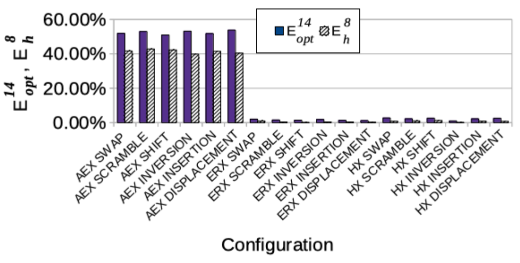
9.2 Adjacency Representation Crossovers and ERX

9.2.1 Methods

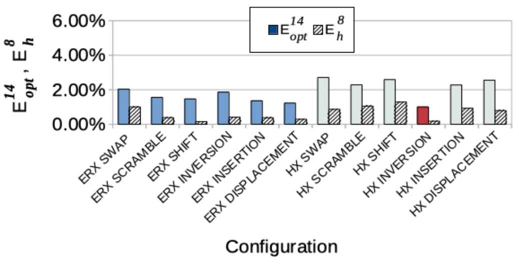
In this part of our experiments, we will compare the effectiveness of adjacency representation crossover operators with each other, namely alternate edges crossover (AEX) and heuristic crossover (HX). Moreover, in addition to them, edge recombination crossover (ERX) will be considered in these experiments as well for the following reason: Although it does not work with the adjacency representation, as it was mentioned in Section 6.4, it also aims at preserving edges from the parents which is relevant for the TSP. Therefore, this feature can cause better performance of these crossover operators in comparison to the path representation crossovers.

In addition, we combine these 3 crossover types with 6 different mutation types, namely swap, shift, scramble, inversion, insertion, and displacement. As a result, we get 18 configurations in total.

9.2.2 Results and Discussion



(a) AEX, ERX, and HX



(b) ERX and HX

Figure 48: Relative error for instances with less than 100 cities.

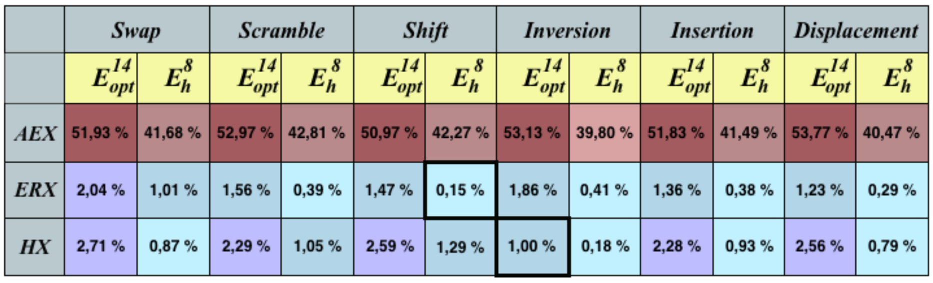
In the group of TSP instances with less than 100 cities (see Figure 48a), the AEX has shown much worse results than the ERX and the HX. Its bad performance in our experiments confirms the results reported in the literature (see [11]) and can be easily explained by the fact that the AEX simply alternately takes the edges from the parents.

If we compare the ERX and the HX in detail (see Figure 48b), we can see that the HX combined with the inversion mutation gives the best results. Please note that for small instances of this group which have less than 30 cities, our genetic algorithm has found the optimal value in many configurations.

It is not surprising that for this group of instances with less than 100 cities, the HX performed much better than the AEX: The HX works with the information about the distances between the cities which is extremely relevant for the TSP. However, the fact that the HX did not perform considerably better than the ERX is rather surprising. As it was explained in Section 6.3, the ERX does not uses the domain information like the HX does. It just iteratively takes an edge from the parents which has the smallest number of neighbors. Probably, this was already enough to come to such good results for small instances. Moreover, as it was mentioned in Section 6.3, the HX tends to repeat the parents if the number of genes is small. Perhaps that is the reason why the HX does not outperform the ERX on the small instances.

As the winner combination for this group of instances, we choose the HX with inversion as it has shown the best results with respect to E_{opt}. Please note that the ERX with shift mutation has shown slightly better results concerning E_h. However, the number of instances which were used for computing E_h is considerably smaller than the number of instances which were used for computing E_{opt}. We thus assume that the results with respect to E_{opt} are more stable.

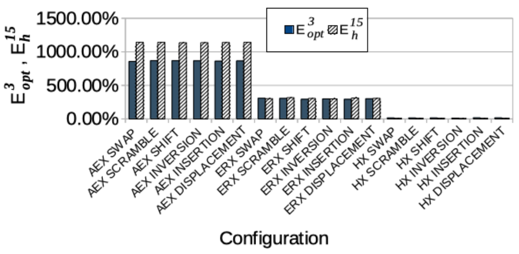
If we take a look at Table 7, there is no clear tendency concerning a certain mutation type. One can say that the shift, inversion and displacement mutation operators perform slightly better than the others. However, the crossover operators show here much stronger influence on the results. It can be easily explained by the fact that these mutation operators were not explicitly developed for the adjacency representation. They do not preserve edges from the parents and make rather random changes. One can therefore make an assumption that applying these mutation operators here can even make the results worse. However, we have conducted a small experiment where we did not use the mutation with the adjacency operators at all: The results without applying mutation were worse than when the mutation takes place. Therefore, we can conclude that mutation is a very important component of genetic algorithms even if one uses the mutation operators which are not perfectly suited for the chosen crossover operators.



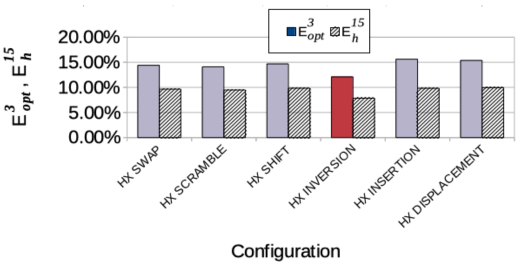
	Swap		Scramble		Shift		Inversion		Insertion		Displacement	
	E_{opt}^{14}	E_h^8	E_{opt}^{14}	E_h^8	E_{opt}^{14}	E_h^8	E_{opt}^{14}	E_h^8	E_{opt}^{14}	E_h^8	E_{opt}^{14}	E_h^8
AEX	51,93 %	41,68 %	52,97 %	42,81 %	50,97 %	42,27 %	53,13 %	39,80 %	51,83 %	41,49 %	53,77 %	40,47 %
ERX	2,04 %	1,01 %	1,56 %	0,39 %	1,47 %	0,15 %	1,86 %	0,41 %	1,36 %	0,38 %	1,23 %	0,29 %
HX	2,71 %	0,87 %	2,29 %	1,05 %	2,59 %	1,29 %	1,00 %	0,18 %	2,28 %	0,93 %	2,56 %	0,79 %

Table 7: Results for 22 instances with less than 100 cities for AEX, ERX, HX operators and 6 mutation operators.

Now let us take a look at the results which were obtained for the TSP instances with 200 to 500 cities (see Figure 49a). One can see here a significant difference among the three crossover operators: The AEX performs worst again, the ERX shows better results, but the definite winner is the HX again. The middle results of the ERX here correspond more to our expectations than the results for smaller instances. The



(a) AEX, ERX, and HX



(b) HX

Figure 49: Relative error for instances with 200 to 500 cities.

outperformance of the HX is not surprising as well because this operator uses the information which is very relevant for the TSP. Therefore, it is better applicable to the TSP problem. If we take a look at this crossover type more in detail, than we can see that the combination with the inversion mutation is slightly better than the others (see Figure 49b).

	Swap		*Scramble*		*Shift*		*Inversion*		*Insertion*		*Displacement*	
	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h	E^3_{opt}	E^{15}_h
AEX	856,90 %	1142,12 %	869,03 %	1141,99 %	868,22 %	1135,99 %	866,72 %	1137,44 %	861,50 %	1139,43 %	865,14 %	1141,70 %
ERX	308,12 %	301,71 %	306,08 %	316,80 %	293,16 %	305,07 %	298,95 %	298,82 %	292,32 %	311,33 %	300,30 %	306,39 %
HX	14,40 %	9,64 %	14,09 %	9,48 %	14,68 %	9,83 %	12,09 %	7,85 %	15,62 %	9,81 %	15,36 %	9,95 %

Table 8: Results for 18 instances with 200 to 500 cities for AEX, ERX, HX operators and 6 mutation operators.

If we take a look at Table 8, we again observe that the mutation operator does not seem to influence the results. No mutation type can be found which has consistently better results than the other. Here, the influence of crossover types is significantly stronger.

Summarizing the results of this part of our experiments, the HX in combination with the inversion mutation has performed best for both groups of instances and is considered as the winner for the adjacency representation crossovers.

9.3 Ordinal Representation with OPX

9.3.1 Methods

For the ordinal representation, we consider the classical one-point crossover (OPX) (see Section 6.1). It was claimed [21] that this representation type is only of historic interest because it has shown very bad results. The purpose of this experiment is to validate this claim on our dataset. Moreover, we will combine this crossover type with 6 different mutation types, namely swap, shift, scramble, inversion, insertion, and displacement. As a result, we get 6 configurations in total.

9.3.2 Results and Discussion

In the group of TSP instances with less than 100 cities, the OPX has shown the best results in combination with the inversion mutation and the worst ones with the scramble mutation (see Figure 50).

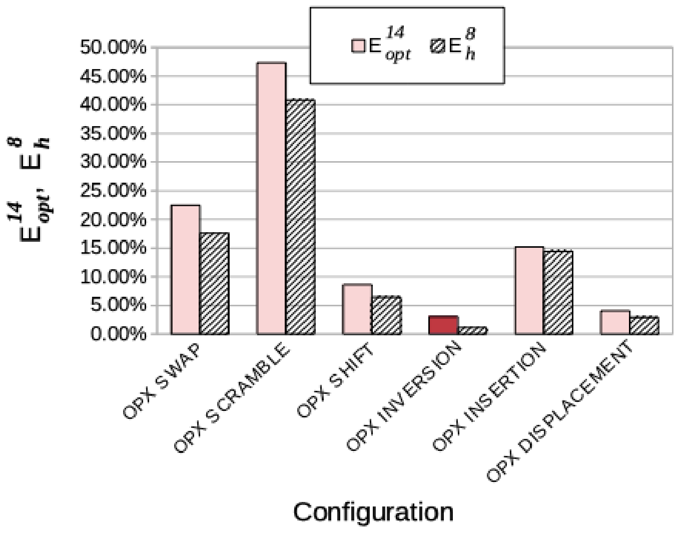


Figure 50: Results for 22 instances with less than 100 cities for the OPX and 6 mutation operators.

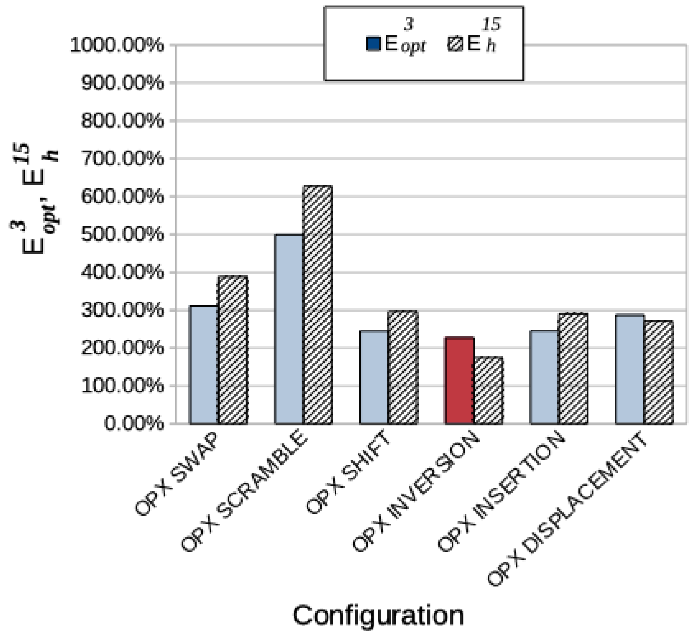


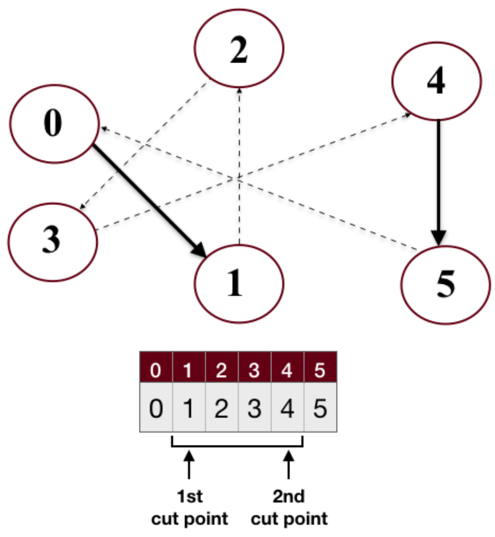
Figure 51: Results for 18 instances with 200 to 500 cities for the OPX and 6 mutation operators.

In the group with larger instances with 200 to 500 cities, the OPX has also shown the best results combined with the inversion mutation and the worst ones with the scramble mutation (see Figure 51).

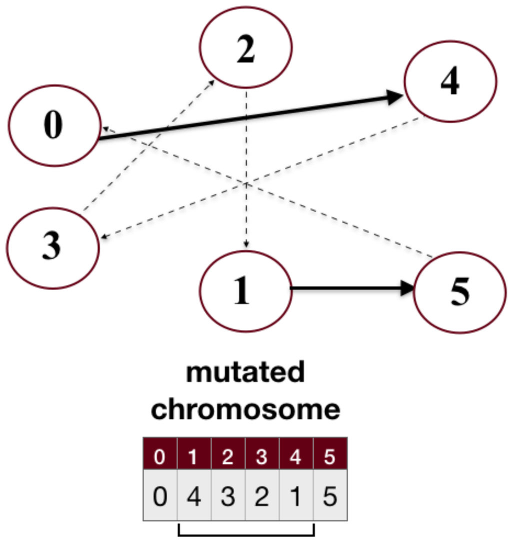
Whether this crossover type can compete with the crossover operators of other representation types, will be validated in the next section, where we are going to compare the results among the representation types.

9.4 Best and worst mutation operators

Summarizing the results of the previous sections, we can definitely state that the inversion mutation has always shown the best results for all crossover operators while the scramble mutation has performed poorly. A great importance of inversion was also described by Potvin [21]. One can explain it by the fact that the inversion mutation changes only two edges preserving the other ones (see Section 7). It changes only the edges which stand at the start and the end of the interval where the inversion happens. Strictly speaking, the other edges can be changed as well but these changes may affect only the direction. However, a change in direction does not influence the length of the tour in symmetric TSP instances, and it does not therefore affect the fitness value.



(a) Graph for the chromosome encoding the tour 012345.

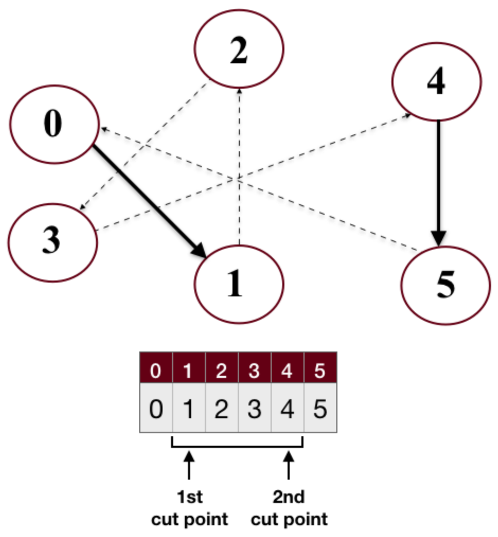


(b) Applying the inversion mutation with the cut points 1 and 4

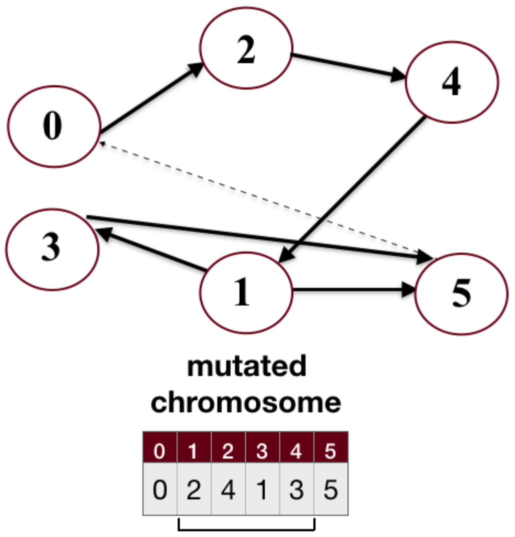
Figure 52: Changes in the graph while applying the inversion mutation with the cut points 1 and 4.

For instance, consider the chromosome which encodes the tour 012345 in path representation. Let us assume that the cut points are 1 and 4 between which the inversion will happen (see Figure 52a). This means that the edge (0,1) stands at the beginning of the interval and the edge (4, 5) stands at the end of it. Applying the inversion here removes these two edges and produces two new edges (0, 4) and (1, 5) (see Figure 52b).

On the other side, applying a scramble mutation to the same chromosome with the same cut points can result into destroying almost all edges (see Figure 53). These changes can therefore completely destroy the results of the crossover operators.



(a) Graph for the chromosome encoding the tour 012345.



(b) Applying the scramble mutation with the cut points 1 and 4.

Figure 53: Changes in the graph while applying the scramble mutation with the cut points 1 and 4.

10 Comparison of Representation Types

10.1 Methods

The aim of this chapter is to compare the performance among the different representation types. At first, we are going to take a look at all results from the previous chapters to find out whether any of the representation types is clearly superior or inferior to the others. This includes investigating whether the ordinal representation always performs worst as it was stated by Potvin [21].

After that, we will consider the winner crossover operators in each group which were determined in Chapter 9. For each of them, we will separately tune the hyperparameters using the inversion mutation as the winner mutation type. This time we will conduct a deeper parameter search considering the following combinations: $s_1 = 3$, and $s_2 = 5$ for the population size factor. In addition, we will consider two variants for the fixed size of population $size_1 = 100$, and $size_2 = 200$ to find out whether our approach of multiplying the length of the tour with some factor for population initialization performs better than just using a fixed size. Moreover, three mutation rates are used, namely $m_1 = 3\%$, $m_2 = 5\%$, and $m_3 = 10\%$. Therefore, we have 12 combinations which will be averaged over 3 seeds. It is important to mention that this deeper parameter search will be conducted twice. First, for the random initialization of the population and then for the population initialized with the help of heuristics. These experiments will be conducted on the two groups of instances as before, namely on the instances with less than 100 cities, and the ones which have from 200 to 500 cities. For each crossover type, we will choose two best combinations of hyperparameters, and then we will use the third dataset with 100 to 200 cities to find the best configuration.

Having defined the best configuration of hyperparameters for each crossover type, we will launch them on all five groups of TSP instances and determine the overall winner. We will conduct this last experiment first using heuristics to find the best result these operators can achieve within 10 minutes. After that, we will use only random permutations to initialize the population to take a look whether the genetic algorithm can outperform the results of construction heuristics.

10.2 Results and Discussion

If we compare the results of all groups of crossover operators (see Figure 54 and Figure 55), we can see that no representation type has performed constantly better or worse than the other ones: In each group, some operators have shown very good results, and some operators have performed poorly.

As mentioned before, Potvin [21] has claimed that the ordinal representation cannot perform well enough to compete with other representation types. However, in our experiments, this representation type has definitely not performed worst. On our dataset which in this experiment included 40 instances, the OPX has outperformed the OBX which has shown the poorest results among the path representation crossovers and the

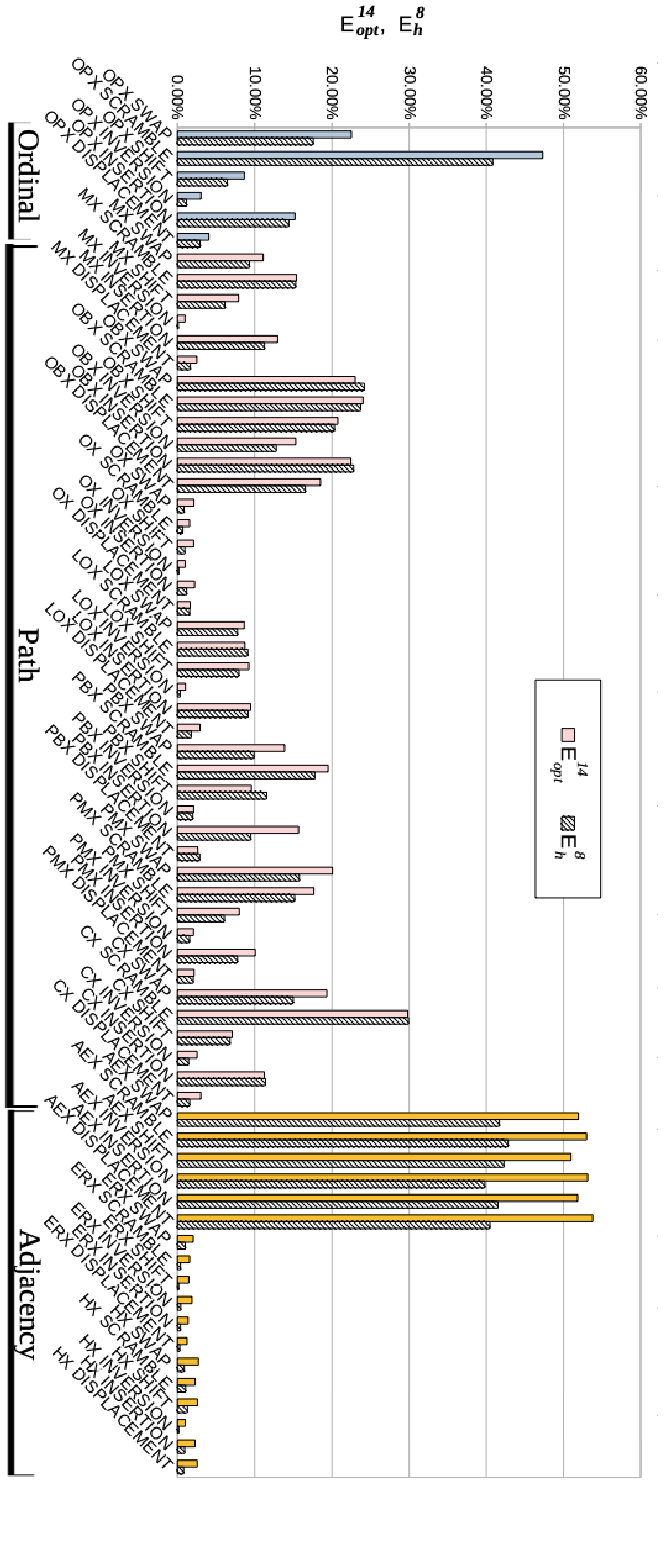


Figure 54: Results for 22 instances with less than 100 cities for different representation types crossovers combined with 6 mutation operators.

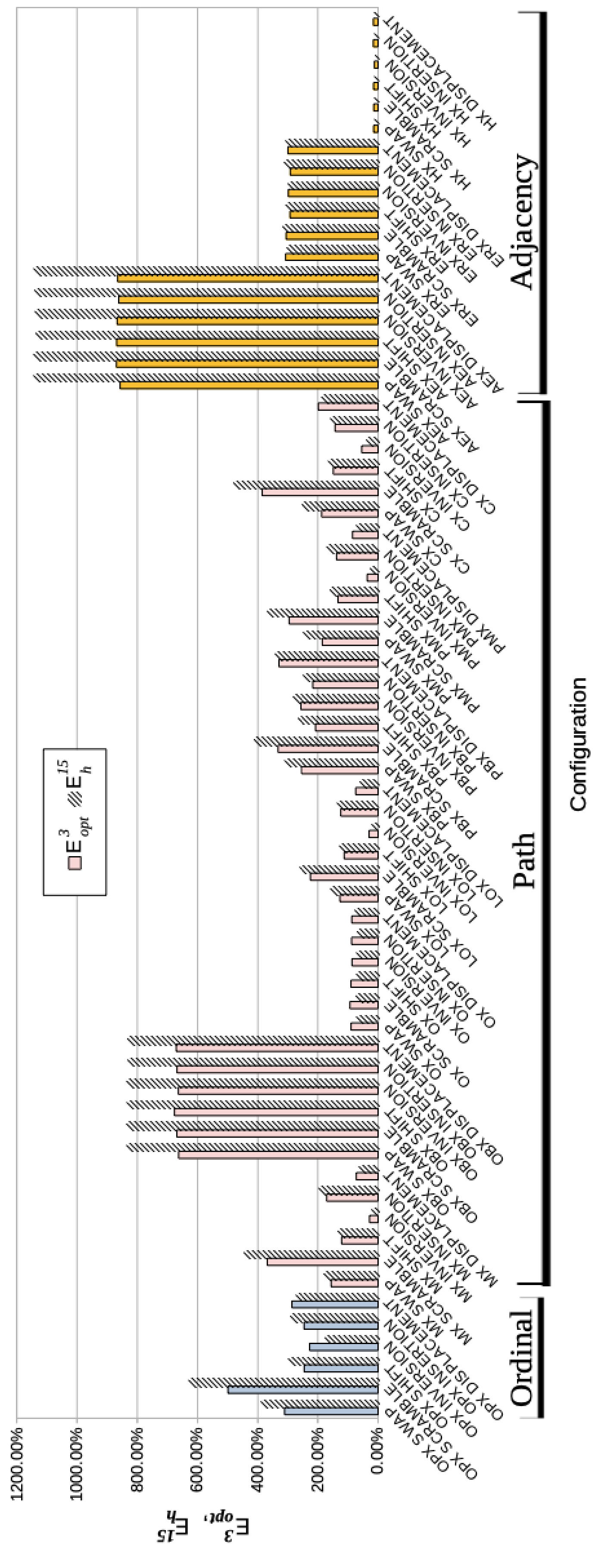


Figure 55: Results for 18 instances with 200 to 500 cities for different representation types crossovers combined with 6 mutation operators.

AEX which has shown the poorest results in the group of the adjacency representation crossovers. However, it could not compete with the crossover operators which have shown the best results, namely the MX, the LOX, the OX, and the HX. Therefore, one can not exclude that the ordinal representation can perform well if, for instance, some specific crossover and mutation operators are developed for it. The ordinal representation always produces valid offsprings. It therefore does not need additional steps to ensure the validity for the TSP as the path and ordinal representation types do (see Section 3.2). This fact makes this representation type more generally applicable for other combinatorial optimization problems. Therefore, perhaps, the ordinal representation is worth being developed further.

If we take a look at the experiments conducted by Starkweather et al. [27], we can see that in addition to 5 crossover operators of the path representation types (OX, OBX, PBX, PMX, and CX), they launched the ERX as well. This operator has shown much better results in their experiments having achieved the optimum in all 30 trials. Please note that these studies used a single small instance with 30 cities. If we take a look at our experiments for the instances with less that 100 cities (see Figure 54), we can confirm that the ERX performed better than or comparable to the path representation crossovers.

However, we cannot confirm the claim made by Potvin [21] that "the edge-preserving operators are superior to the other types of crossover operators". His claim is based on the studies made by Starkweather et al. [27] who considered only ERX using only one small instance with 30 cities and on the studies made by Liepins et al. [18] which used only two types of crossover operators. In our experiment, the AEX performed much worse than all other operators of all groups despite the fact that it is an edge-preserving operator. The ERX has shown a poor performance for instances with 200 to 500 cities as well. Only the HX has performed very good among the edge preserving operators.

Now we are going to tune the hyperparameters for the operators which have performed best in the previous experiments, namely the HX, the LOX, the OX, and the MX all combined with the inversion mutation (see Chapter 9). At first, we have conducted the experiments without using heuristics to initialize the population. It is interesting that the configuration of hyperparameters where the population size was fixed at 100 chromosomes with the mutation rate 10% has performed best for most operators (see Table 9). However, there are still many different configurations which perform best for a specific operator and a specific group of instances. Therefore, we conclude that the best configuration of hyperparameters seems to be very sensitive to the size of a TSP instance and the operators used.

Now let us take a look at the results, where the construction heuristics were used to initialize the population (see Table 10). Please note that we use the results of heuristics as a benchmark for the instances with no known optimal value. In case of initialization with heuristics, our genetic algorithm can, of course, only outperform the results made by heuristics. Therefore, we will speak about the improvement of the genetic algorithm with respect to the heuristics in this case which we want to maximize (see Figure 10b). Here, we can see even higher differentiation concerning the best configuration. This

		< 100 cities								200 to 500 cities							
		MX		LOX		OX		HX		MX		LOX		OX		HX	
s/ size	m	E_{opt}^{14}	E_h^8	E_{opt}^{14}	E_h^8	E_{opt}^{14}	E_h^8	E_{opt}^{14}	E_h^8	E_{opt}^3	E_h^{15}	E_{opt}^3	E_h^{15}	E_{opt}^3	E_h^{15}	E_{opt}^3	E_h^{15}
3	3 %	1,11 %	0,42 %	1,22 %	0,57 %	2,60 %	1,21 %	1,07 %	0,18 %	31,30 %	21,95 %	30,37 %	15,91 %	101,45 %	78,28 %	12,09 %	7,90 %
3	5 %	1,22 %	-0,23 %	1,58 %	0,78 %	3,11 %	1,68 %	0,82 %	-0,06 %	26,56 %	15,32 %	25,23 %	11,34 %	97,91 %	80,07 %	12,52 %	8,01 %
3	10 %	0,66 %	-0,13 %	1,19 %	0,14 %	2,83 %	1,70 %	0,79 %	-0,06 %	19,35 %	10,77 %	24,49 %	9,86 %	99,28 %	78,99 %	11,94 %	7,04 %
5	3 %	1,27 %	-0,65 %	1,05 %	0,65 %	3,07 %	1,84 %	1,21 %	0,13 %	71,14 %	51,48 %	62,58 %	37,23 %	99,66 %	80,34 %	13,99 %	8,14 %
5	5 %	1,30 %	-0,82 %	0,99 %	0,86 %	2,85 %	1,46 %	0,89 %	-0,13 %	61,36 %	41,79 %	52,84 %	30,25 %	103,42 %	83,39 %	13,77 %	7,79 %
5	10 %	0,74 %	0,02 %	1,02 %	0,52 %	2,74 %	1,37 %	0,72 %	-0,03 %	48,28 %	34,35 %	54,58 %	26,42 %	97,65 %	89,70 %	12,32 %	6,85 %
100	3 %	0,79 %	-0,46 %	1,35 %	0,46 %	1,10 %	0,03 %	0,95 %	-0,07 %	8,48 %	3,74 %	8,95 %	3,18 %	23,47 %	16,14 %	12,35 %	6,02 %
100	5 %	0,85 %	-0,31 %	1,22 %	0,58 %	1,25 %	0,21 %	0,64 %	-0,06 %	9,22 %	2,72 %	8,16 %	2,03 %	24,04 %	15,04 %	10,97 %	5,75 %
100	10 %	0,77 %	-0,06 %	1,10 %	0,36 %	0,94 %	0,37 %	0,47 %	-0,38 %	7,91 %	2,45 %	7,47 %	2,05 %	22,69 %	12,40 %	11,49 %	5,46 %
200	3 %	1,44 %	-0,36 %	1,02 %	0,19 %	1,03 %	0,23 %	0,95 %	-0,21 %	10,26 %	4,38 %	10,23 %	3,24 %	32,90 %	23,04 %	11,95 %	7,14 %
200	5 %	1,17 %	-0,39 %	1,44 %	0,49 %	0,95 %	0,28 %	0,91 %	0,06 %	7,97 %	3,55 %	7,81 %	2,60 %	32,76 %	22,65 %	12,84 %	6,18 %
200	10 %	0,89 %	0,32 %	1,19 %	-0,21 %	1,38 %	0,52 %	0,72 %	-0,10 %	7,68 %	2,63 %	8,11 %	1,85 %	37,61 %	23,10 %	12,31 %	6,17 %

Table 9: Results for 12 configurations for the MX, LOX, OX, and HX for the random initialization of population, all combined with the inversion mutation.

		< 100 cities				200 to 500 cities			
		MX	LOX	OX	HX	MX	LOX	OX	HX
s/ size	m	E_{opt}^{14}	E_{opt}^{14}	E_{opt}^{14}	E_{opt}^{14}	E_{opt}^3	E_{opt}^3	E_{opt}^3	E_{opt}^3
3	3 %	0,44 %	0,40 %	0,45 %	0,47 %	6,12 %	6,42 %	6,38 %	7,77 %
3	5 %	0,47 %	0,50 %	0,48 %	0,59 %	5,94 %	5,97 %	6,48 %	7,40 %
3	10 %	0,39 %	0,43 %	0,47 %	0,50 %	5,70 %	6,20 %	6,56 %	7,67 %
5	3 %	0,45 %	0,44 %	0,49 %	0,59 %	6,10 %	6,51 %	6,25 %	7,70 %
5	5 %	0,42 %	0,43 %	0,43 %	0,48 %	5,63 %	6,33 %	6,50 %	7,77 %
5	10 %	0,46 %	0,45 %	0,50 %	0,51 %	5,80 %	6,55 %	6,42 %	7,77 %
100	3 %	0,39 %	0,43 %	0,48 %	0,57 %	6,39 %	5,99 %	6,34 %	7,74 %
100	5 %	0,40 %	0,42 %	0,46 %	0,52 %	5,79 %	6,23 %	6,61 %	7,61 %
100	10 %	0,40 %	0,50 %	0,46 %	0,50 %	5,84 %	6,07 %	6,75 %	7,31 %
200	3 %	0,41 %	0,40 %	0,48 %	0,59 %	6,38 %	5,99 %	6,45 %	7,77 %
200	5 %	0,40 %	0,40 %	0,48 %	0,49 %	5,79 %	6,15 %	6,61 %	7,61 %
200	10 %	0,40 %	0,44 %	0,45 %	0,52 %	5,84 %	6,06 %	6,75 %	7,31 %

(a) Relative error with respect to the optimal value averaged over 14 instances.

		< 100 cities				200 to 500 cities			
		MX	LOX	OX	HX	MX	LOX	OX	HX
s/ size	m	I_h^8	I_h^8	I_h^8	I_h^8	I_h^{15}	I_h^{15}	I_h^{15}	I_h^{15}
3	3 %	0,40 %	0,32 %	0,42 %	0,20 %	1,09 %	1,08 %	1,01 %	0,10 %
3	5 %	0,39 %	0,29 %	0,40 %	0,26 %	1,22 %	1,18 %	1,00 %	0,11 %
3	10 %	0,39 %	0,37 %	0,43 %	0,26 %	1,40 %	1,34 %	0,95 %	0,21 %
5	3 %	0,38 %	0,30 %	0,44 %	0,29 %	1,20 %	0,99 %	0,90 %	0,12 %
5	5 %	0,31 %	0,39 %	0,39 %	0,28 %	1,05 %	1,04 %	1,00 %	0,20 %
5	10 %	0,38 %	0,35 %	0,50 %	0,22 %	1,33 %	1,14 %	0,81 %	0,20 %
100	3 %	0,35 %	0,28 %	0,38 %	0,11 %	1,19 %	1,27 %	1,00 %	0,15 %
100	5 %	0,34 %	0,35 %	0,44 %	0,25 %	1,20 %	1,32 %	0,94 %	0,09 %
100	10 %	0,40 %	0,41 %	0,44 %	0,26 %	1,38 %	1,42 %	1,06 %	0,15 %
200	3 %	0,33 %	0,39 %	0,44 %	0,25 %	1,18 %	1,27 %	1,00 %	0,15 %
200	5 %	0,39 %	0,35 %	0,46 %	0,25 %	1,20 %	1,31 %	0,94 %	0,09 %
200	10 %	0,46 %	0,37 %	0,50 %	0,23 %	1,38 %	1,42 %	1,05 %	0,15 %

(b) Relative improvement with respect to the best results of heuristics (with ten minutes limit) averaged over 8 instances.

Table 10: Results for 12 configurations for the MX, LOX, OX, and HX for the initialization of population with heuristics, all combined with the inversion mutation.

supports our point about the sensibility of the hyperparameters to the size of a TSP instance and the used operators.

Moreover, we can see that the best configuration of parameters for the same operators differs if different types of initializing the population are used (see Tables 9 and 10). This illustrates that the hyperparameters are sensible not only to the number of cities and the type of operators but to the way how the population is initialized as well.

	Random initialization		*Initialization with heuristics*		*Configurations for further search*
	< 100 cities	*200 to 500 cities*	*< 100 cities*	*200 to 500 cities*	
	Configuration (number of cities over which the average is built)				
MX	**3/10% (14), 5/5% (8)**	**100/10% (18)**	**3/10% (14), 100/3% (14), 200/10% (8)**	**5/5% (3), 3/10% (15)**	**3/10%, 100/10%**
OX	**100/3% (8), 100/10% (14)**	**100/10% (18)**	**5/5% (14), 5/10% (8), 200/10 (8)%**	**5/3% (3), 100/10% (15)**	**100/10%, 5/5%**
LOX	**5/5% (14), 200/10% (8)**	**100/10% (3), 200/10% (15)**	**3/3% (14), 200/3% (14), 200/5% (14), 100/10% (8)**	**3/5% (3), 100/10% (15), 200/10% (15)**	**100/10%, 200/10%**
HX	**100/10% (22)**	**100/5% (3), 100/10% (15)**	**3/3% (14), 5/3% (8)**	**3/10% (15), 100/10% (3), 200/10% (3)**	**3/10%, 100/10 %**

Table 11: Best configurations for the MX, LOX, OX, and HX which are all combined with the inversion mutation.

100 to 200 cities	*Population size/ Mutation rate* (E^{9}_{opt})		*Population size/ Mutation rate* (E^{15}_{h})		**Winner configuration**
MX	**100/10% (2,64%)**	3/10% (3,32%)	**100/10% (0,46%)**	3/10% (0,78%)	100/10%
OX	**100/10%(2,78%)**	5/5% (3,90)	**100/10% (0,88%)**	5/5% (3,07%)	100/10%
LOX	**100/10% (2,81%)**	200/10% (2,92%)	**100/10% (-0,01%)**	200/10% (0,67%)	100/10%
HX	**100/10 % (3,45)**	3/10% (3,72%)	**100/10% (1,83%)**	3/10% (2,24%)	100/10%

Table 12: Winner configurations for the MX, LOX, OX, and HX for random initialization which are all combined with the inversion mutation.

100 to 200 cities	*Population size/ Mutation rate* (E^{9}_{opt})		*Population size/ Mutation rate* (I^{15}_{h})		**Winner configuration**
MX	**100/10% (1,13%)**	3/10% (1,17%)	100/10% (1,00%)	**3/10% (1,01%)**	100/10%, 3/10%
OX	**100/10%(1,16%)**	5/5% (1,22)	100/10% (0,91%)	**5/5% (1,06%)**	100/10%, 5/5%
LOX	100/10% (1,03%)	200/10% (1,03%)	100/10% (1,07%)	200/10% (1,07%)	100/10%, 200/10%
HX	**100/10 % (1,87%)**	3/10% (2%)	**100/10% (0,42%)**	3/10% (0,39%)	100/10%

Table 13: Winner configurations for the MX, LOX, OX, and HX for initialization with heuristics which are all combined with the inversion mutation.

The best configurations for each crossover operator are given in Table 11. Among them, for each crossover operator, we choose the two combinations which have shown the best results for more instances. If two configurations were the best for the same number of instances, we randomly take one of them. All these chosen configurations were tested on another group of instances which have 100 to 200 cities. The results are given in Tables 12 and 13. As one can see, the configuration with the fixed population size at 100 and the mutation rate 10 has performed best for all four operators.

Now we launch the experiments on the remaining two groups of instances, namely the one with 500 to 1500 cities and the one with 1500 to 10000 cities using the selected configurations, namely population size fixed at 100 and the mutation rate of 10% for all four crossover operators. We will get thus the results for the whole dataset.

At first we conducted the experiments without using heuristics to initialize the population. We have to use two benchmarks in our experiments because some instances do not have a known optimal value. As we have five groups of instances, we therefore get 10 charts with the results (see 56). For each operator, we calculate the number of charts where it has shown the best results. Overall, the HX has outperformed in 5 subgroups of instances which we can see in the 5 corresponding charts, the LOX is best in 4, the MX has outperformed in 1 case, and the OX was better in none og the subgroups. Of course, some subgroups have very small number of instances, as for instance, there is only one instance with a known optimal value in the group of instances with 1500 to 10000 cities. So the result of E^1_{opt} here can be considered as outlier. Therefore, let us calculate instead the number of instances for which the corresponding operator has shown the best averaged results HX: $14 + 1 + 8 + 20 + 14 = 57$, LOX: $3 + 3 + 15 + 15 = 36$, MX: 9 ,and OX: 0.

Thus, the HX has shown the best results on average independent from the benchmark we used. This is not surprising because the population here includes only random permutations of cities which are more likely to be bad tours. The HX chooses iteratively the shortest edge from the parents which is a very relevant piece of information for the TSP and as a result can significantly improve the initial bad tours. Moreover, the fact that the HX tends to repeat parents for small instances does not seem to impair its performance.

For the instances with number of cities smaller than 200, we can observe negative values of E_h which corresponds to an improvement over the heuristics. The genetic algorithm, where the population was initialized only with random permutations, could thus outperform the heuristics despite the relatively short time given, namely ten minutes. Of course, we can see the drastic decrease in performance with the increase in dimension. However, it is important to mention that a stop criterion of ten minutes resulted in much greater number of iterations for the smaller instances than for the larger ones. This means that the genetic algorithm does not necessarily perform much worse on larger instances than on the smaller ones if it gets sufficient time to make the same number of iterations.

It is also important to mention that the results were averaged over the instances across the corresponding group, and the genetic algorithm was able to find the optimal value for some instances which one cannot see in these charts.

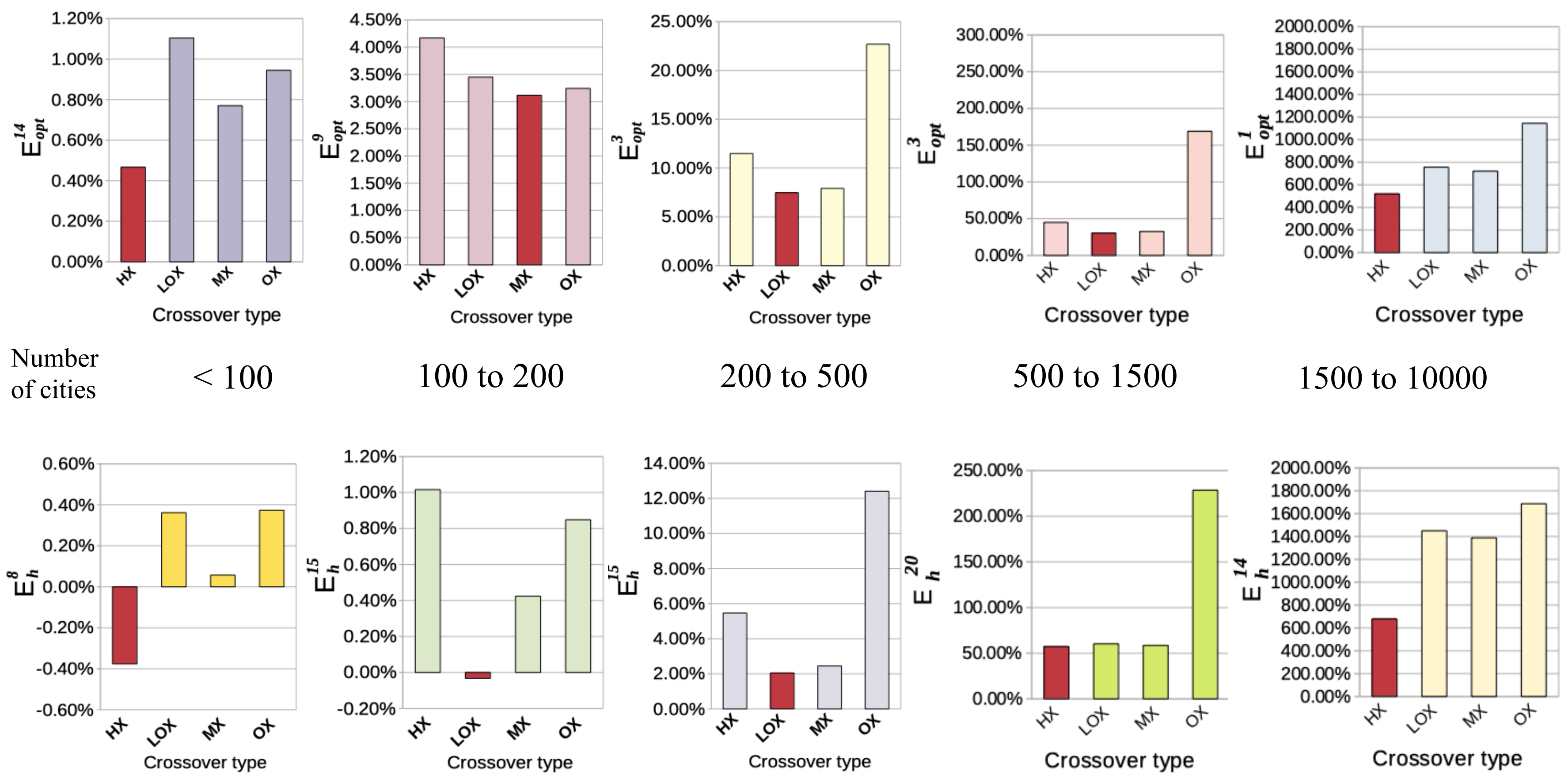


Figure 56: Final results for the MX, LOX, OX, and HX which are all combined with the inversion mutation with the random initialization of population.

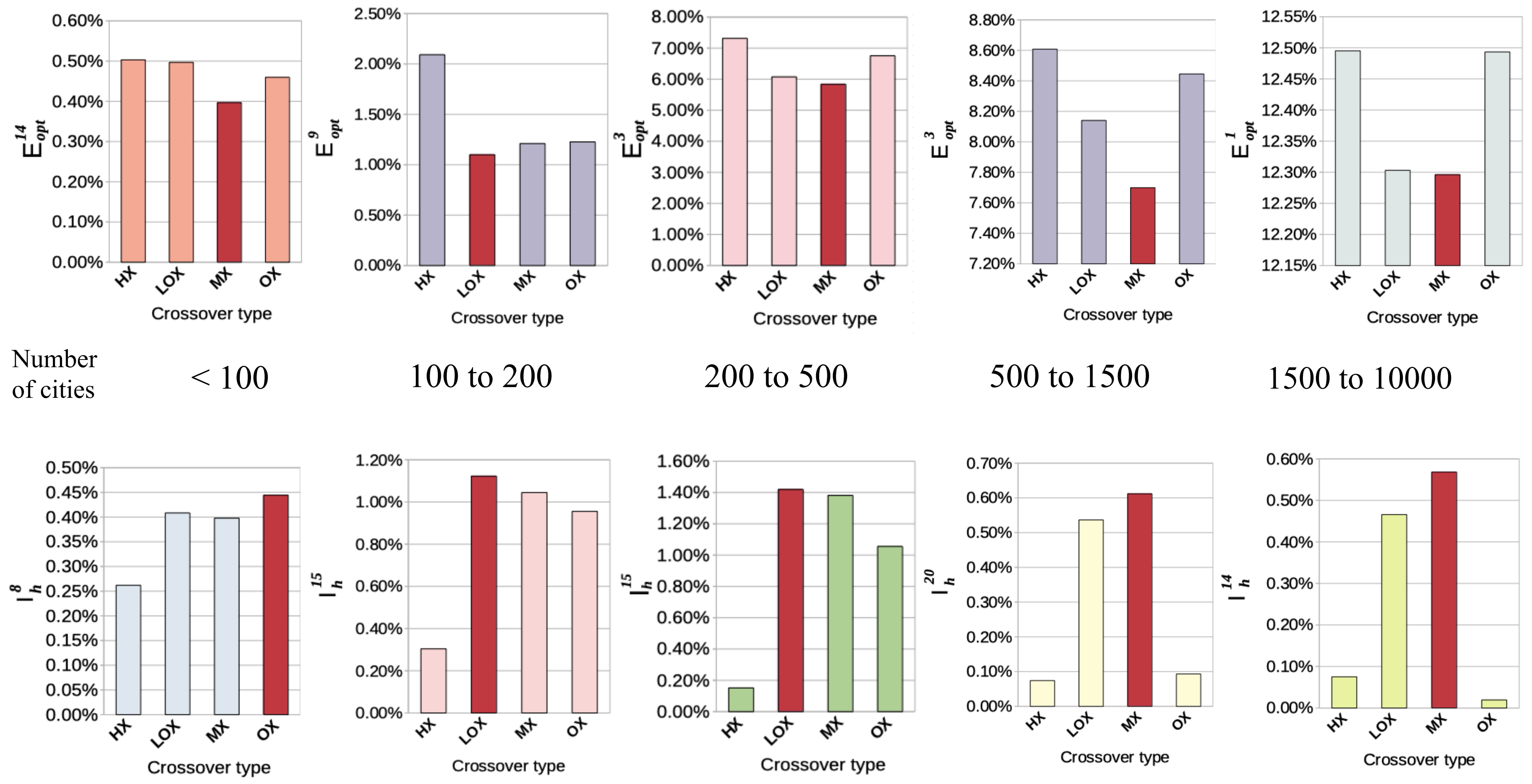


Figure 57: Final results for the MX, LOX, OX, and HX which are all combined with the inversion mutation where the population was initialized using heuristics.

In the next experiments, we used heuristics to initialize the population (see Figure 57). Here, we can notice the decrease in performance with the increase of the number of cities as well. It can be also explained by the fact that for the larger instances, the genetic algorithm could not make sufficient number of iterations in comparison to the smaller instances.

Since the population has been initialized with the results of the construction heuristics, the genetic algorithm can not physically perform worse than the heuristics. Therefore, the relative improvement over the heuristics is considered in this case, and it is maximized.

The MX has outperformed in 7 subgroups of instances which we can see in the 6 corresponding charts, the LOX in 3, the OX in 1, and the HX was best in none of the subgroups. If we calculate the number of instances for which the corresponding operator was on average the best, we have MX: $14 + 3 + 3 + 1 + 20 + 14 = 55$, LOX: $9 + 15 + 15 = 39$, OX: 8, and HX: 0.

It is interesting that the HX here has shown worse results while the MX is the definite winner. As it was described in Section 6.2, the MX makes the simplest changes in comparison to the other crossovers and results therefore in more iterations per time unit. Its great performance here can be explained by the fact that the heuristics used in the population have already come to very good results: The domain information is already used, and the HX cannot offer much in this case but requires more computations and thus more time. The MX, however, makes rather random changes but it makes them much faster. As a result, it slightly improves the initial results of the heuristics.

It is interesting to mention that in these experiments, where the population size was fixed at 100 chromosomes, and the mutation rate was 10%, the OX has performed constantly worse than the LOX. This is exactly the opposite tendency in comparison with the one we reported in Section 9.1, where we multiplied the number of cities by the factor 3 to get the population size, and where the mutation rate was 3%. Therefore, we can conclude that the choice of parameters plays an important role in performance of genetic algorithms.

Considering the two approaches which we used to initialize the population, we can definitely claim that using heuristics in the population improves the performance of the genetic algorithm. For all groups of instances, the results are better when the heuristics are used, and the difference in performance becomes especially visible with the increase of dimension. So, if we work with rather small instances which have less than 100 cities, for instance, the best value with random initialization is $E_{opt}^{14} = 0.47\%$, with the HX used. The best value achieved in this case with using heuristics is $E_{opt}^{14} = 0.40\%$ with the MX. So, the difference here probably does not justify implementing and using the heuristics. The results obtained with the random initialization of the population can be considered as sufficient. But if we work with large instances, the difference is drastic. For instance, if we take a look at the group of instances with 500 to 1500 cities, the best result is $E_{opt}^{3} = 30.36\%$ achieved by the LOX for random initialization while the best relative error for the initialization with heuristics is $E_{opt}^{3} = 7.70\%$ achieved with the MX. Therefore, it definitely makes sense to include the results of heuristics into the initial population if one works with large instances or if even a slightly improvement in performance for smaller instances is important.

11 Conclusion

In the context of this bachelor thesis, a detailed theoretical overview of genetic algorithms in context of the TSP was provided. After that, we have analyzed different combinations of crossover and mutation operators and tuned the parameters for them to find a best variation of a genetic algorithm for solving the TSP. This analysis was based on the results of the experiments which were conducted on about one hundred TSP instances of different size.

The search for the best hyperparameters has shown that a best configuration is very sensitive to the operators used and the size of the TSP instance. Our approach of taking into account the length of the tour when determining the size of the population inspired by the investigations of Chen, Montgomery, and Bolufé-Röhler [7] has not shown any promising results. On average, a population size fixed at 100 chromosomes and a mutation rate of 10% yielded the best results.

Concerning the statement made by Potvin [21] about the path representation crossovers, we can confirm that the crossover operators which preserve relative order perform much better than the crossover operators which preserve absolute order only if the OBX is not taken into consideration.

For small instances with less than 100 cities, our experiments confirm a great performance of the ERX in comparison to other path crossover operators which was shown in studies made by Starkweather et al. [27]. However, we can not confirm the statement by Potvin [21] that "the edge-preserving operators are superior to the other types of crossover operators". On our dataset, no representation type has performed constantly better or worse than the other. Each group of crossover operators has some operators which have performed poorly and has the ones which have shown good results: The OBX was the worst in the group of the path representation crossover operators while the AEX was the worst operator which uses the adjacency representation type. The ordinal representation has not performed worst. However, it has neither shown the best results. This does not support the point of view by Potvin [21] that "this representation is mostly of historic interest".

Concerning the six mutation operators, the inversion mutation has drastically outperformed all the other mutation types in all combinations and for all instances. This confirmed the great importance of this mutation operator declared by Potvin [21]. Among the mutation operators which have shown the worst results, the scramble mutation can be named.

The winner among the crossover operators depends on the way the population is initialized: If only random permutations of cities are used, the HX performs best. If the population is initialized using results of construction heuristics, the MX performs best. The OX crossover has not shown the best results, but it constantly performed well which confirms the results from the literature (see [27]).

The tournament selection performs better if the number of participants is drawn randomly from some interval and not constantly fixed at one number.

Concerning the general performance, our genetic algorithm with random initialization has outperformed the results of four construction heuristics for instances with less than 200 cities. For larger instances, using only the random permutations has performed poorly.

Using the heuristics for population initialization has drastically increased the performance and has shown better results for smaller instances than for the larger ones as well. In fact, the four construction heuristics we used sometimes returned the optimal solution for smaller instances; therefore, using genetic algorithm can be unnecessary in this case.

The fact that the performance of our genetic algorithm was better for the smaller instances than for the larger ones independent from the type of the initialization of the population has proven our point that the length of the tour influences the results. A drastic performance decrease with an increasing problem size in all experiments can be easily explained by the fact that our genetic algorithm has made much more iterations for smaller instances than for the larger ones. As a result, the approach to choose a time limit as a stop criterion is rather questionable if instances of real world applications have to be considered.

Taking into account the fact that our genetic algorithm has made much less iterations for the larger instances, we cannot really evaluate the statement by Potvin [21] that the edge preserving operators (AEX, ERX, and HX) cannot find good solutions for larger TSP instances. Operators of all representation types performed worse in our experiments in this case. In order to evaluate this statement, a fair stop condition like a number of iterations has to be used. Therefore, as the directions for future work, increasing the time limit or using another stop criterion such as number of iterations can be considered.

In our research, we have used four simplest construction heuristics. One can consider other heuristics (e.g., savings heuristic) which can be used to initialize the population. Moreover, the ordinal representation has not shown the worst results; therefore this representation type should not be completely excluded from consideration. Developing different crossover and mutation operators specifically for it could increase its performance. The fact that this representation type always produces valid offsprings without making additional steps to ensure the validity, makes it applicable also to other optimization problems. Finally, concerning the adjacency representation, some special mutation operators which aim at preserving the edges from the fittest parents can be developed for it.

References

[1] B. Akay and X. Yao. Recent advances in evolutionary algorithms for job shop scheduling. In: *Automated Scheduling and Planning*. Springer, 2013, pp. 191–224.

[2] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2006.

[3] J. C. Bean. Genetic algorithms and random keys for sequencing and optimization. In: *Operations Research Society of America (ORSA) Journal on Computing* 6.2 (1994), pp. 154–160.

[4] V. Bechberger. *Genetic algorithms for the TSP, GitHub repository*. https://gitko.informatik.uni-osnabrueck.de/vsakharova/bachelor-thesis. 2020.

[5] R. S. Beed, S. Sarkar, A. Roy, and S. Chatterjee. A Study of the Genetic Algorithm Parameters for Solving Multi-objective Travelling Salesman Problem. In: *Proc. of 2017 International Conference on Information Technology (ICIT)*. 2017, pp. 23–29.

[6] R. Bellman. Dynamic programming treatment of the travelling salesman problem. In: *Journal of the Association for Computing Machinery (JACM)* 9.1 (1962), pp. 61–63.

[7] S. Chen, J. Montgomery, and A. Bolufé-Röhler. Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution. In: *Applied Intelligence* 42.3 (2015), pp. 514–526.

[8] L. Davis. Applying adaptive algorithms to epistatic domains. In: *Proc. of 9th International Conference on Artificial Intelligence (IJCAI)*. 1985, pp. 162–164.

[9] M. Gendreau and J.-Y. Potvin. Metaheuristics in combinatorial optimization. In: *Annals of Operations Research* 140.1 (2005), pp. 189–213.

[10] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In: *Foundations of Genetic Algorithms*. Vol. 1. Elsevier, 1991, pp. 69–93.

[11] J. Grefenstette, R. Gopal, B. Rosmaita, and D. Gucht. Genetic algorithms for the traveling salesman problem. In: *Proc. of 1st International Conference on Genetic Algorithms (ICGA)*. 1985, pp. 160–168.

[12] J. H. Holland. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. In: *The University of Michigan Press, Ann Arbor* (1975).

[13] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier, 2004.

[14] S. Knust. *Lecture notes for Introduction to Combinatorial Optimization*. 2020.

[15] G. Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. In: *European Journal of Operational Research (EJOR)* 59.2 (1992), pp. 231–247.

[16] P. Larranaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. In: *Artificial Intelligence Review* 13.2 (1999), pp. 129–170.

[17] *Library of Traveling Salesman problem instances.* `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html`. Zuse Institute Berlin (ZIB), 1995. (Visited on 05/02/2020).

[18] G. E. Liepins, M. R. Hilliard, J. Richardson, and M. Palmer. Genetic algorithms applications to set covering and traveling salesman problems. In: *Operations research and Artificial Intelligence: The integration of problem-solving strategies.* Springer, 1990, pp. 29–57.

[19] G. E. Liepins, M. R. Hilliard, M. Palmer, and M. Morrow. Greedy genetics. In: *Proc. of 2nd International Conference on Genetic Algorithms (ICGA).* 1987, pp. 90–99.

[20] I. M. Oliver, D. J. Smith, and J. R. C. Holland. Study of permutation crossover operators on the traveling salesman problem. In: *Proc. of 2nd International Conference on Genetic Algorithms (ICGA).* 1987, pp. 224–230.

[21] J.-Y. Potvin. Genetic algorithms for the traveling salesman problem. In: *Annals of Operations Research* 63.3 (1996), pp. 337–370.

[22] N. M. Razali and J. Geraghty. Genetic algorithm performance with different selection strategies in solving TSP. In: *Proc. of International Conference of Computational Intelligence and Intelligent Systems (ICCIIS).* 2011, pp. 1–6.

[23] A. Rexhepi, A. Maxhuni, and A. Dika. Analysis of the impact of parameters values on the Genetic Algorithm for TSP. In: *International Journal of Computer Science Issues (IJCSI)* 10.1 (2013), pp. 158–164.

[24] M. Safe, J. Carballido, I. Ponzoni, and N. Brignole. On stopping criteria for genetic algorithms. In: *Proc. of 17th Brazilian Symposium on Artificial Intelligence (SBIA).* Springer. 2004, pp. 405–413.

[25] P. Sharma and A. Wadhwa. Analysis of selection schemes for solving an optimization problem in genetic algorithm. In: *International Journal of Computer Applications (IJCA)* 93.11 (2014).

[26] L. V. Snyder and M. S. Daskin. A random-key genetic algorithm for the generalized traveling salesman problem. In: *European Journal of Operational Research (EJOR)* 174.1 (2006), pp. 38–53.

[27] T. Starkweather, S. McDaniel, K. E. Mathias, L. D. Whitley, and C. Whitley. A Comparison of Genetic Sequencing Operators. In: *Proc. of 4th International Conference on Genetic Algorithms (ICGA).* 1991, pp. 69–76.

[28] J. Y. Suh and D. Van Gucht. Incorporating Heuristic Information into Genetic Search. In: *Proc. of 2nd International Conference on Genetic Algorithms (ICGA).* 1987, pp. 100–106.

[29] L. D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In: *Proc. of 3rd International Conference on Genetic Algorithms (ICGA).* 1989, pp. 133–40.

[30] Y. Zhang, Q. Ma, M. Sakamoto, and H. Furutani. Effects of population size on the performance of genetic algorithms and the role of crossover. In: *Artificial Life and Robotics* 15.2 (2010), pp. 239–243.

Erklärung zur selbständigen Abfassung der Bachelorarbeit

Name: Bechberger, Valeriya

Geburtsdatum: 14.07.1988

Matrikelnummer: 967112

Titel der Bachelorarbeit: Genetic Algorithms for the Traveling Salesman Problem

Ich versichere, dass ich die eingereichte Bachelor thesis selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommene Stellen habe ich kenntlich gemacht.

Osnabrück, June 10, 2020

____

(Valeriya Bechberger)